



NEW YORK COLLEGE
THE INTERNATIONAL COLLEGE OF GREECE



Samouil Mosios

SM120MPO

BSc (Hons) Computing

Dissertation:

**Leveraging MLOps and Continuous
Learning for Predictive Analysis of Traffic
Accidents in the UK**

Supervisor: Stylianos Kokkas

January 2025

Abstract

Traffic accidents pose significant societal and economic challenges, particularly due to their unpredictability and potential severity. Accurate severity classification is essential for improving resource allocation, enabling prompt emergency responses, and enhancing road safety. This thesis explores the integration of MLOps methodologies to develop a continuous learning pipeline for predictive analysis of traffic accident severity in England. Leveraging approximately 660,000 historical accident records with a focus on the three severity classes - Slight, Serious, and Fatal - the study addresses the prevalent class imbalance where Serious and Fatal cases constitute 77,000 instances. By utilizing the MLRun framework for orchestration and automation, the pipeline automates the continuous learning of machine learning models with incoming batch data. This approach ensures the models remain adaptive to evolving traffic conditions. The research emphasizes the integration of DevOps principles within MLOps frameworks, showcasing how continuous learning and automation enhance model reliability. This work contributes to the growing field of automated predictive analytics by demonstrating how leveraging MLOps can improve the quality, scalability and maintainability of machine learning models in real-world scenarios. The findings highlight the potential of continuous learning pipelines to mitigate the impact of traffic accidents, offering valuable insights for policymakers, urban planners, and emergency services.

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 1.1 | Background | 1 |
| 1.2 | Research problem | 1 |
| 1.3 | Aims and objectives | 2 |
| 1.4 | Research questions | 3 |
| 1.5 | Theoretical framework | 3 |
| 1.6 | Research significance | 3 |
| 1.7 | Scope and limitations | 4 |
| 1.8 | Dissertation outline | 4 |
| 2 | Literature Review | 6 |
| 2.1 | Traffic accidents analysis | 6 |
| 2.1.1 | Spatiotemporal hotspots | 6 |
| 2.1.2 | Accident severity classification | 7 |
| 2.2 | Machine Learning | 8 |
| 2.2.1 | Data Analysis | 8 |
| 2.2.2 | ML models | 9 |
| 2.3 | Contemporary Operations | 12 |
| 2.3.1 | DevOps | 12 |
| 2.3.2 | MLOps | 14 |
| 2.4 | ML in traffic accident prediction | 17 |
| 2.5 | Research Gaps | 18 |

| | | |
|----------|---|-----------|
| 3 | Methodology | 20 |
| 3.1 | Introduction | 20 |
| 3.2 | System Architecture | 20 |
| 3.2.1 | Infrastructure Design | 21 |
| 3.2.2 | Source code to Kubernetes job | 23 |
| 3.2.3 | Continuous Integration | 24 |
| 3.3 | Data collection and preprocessing | 31 |
| 3.3.1 | Data Loading | 31 |
| 3.3.2 | Initial Data Exploration | 31 |
| 3.3.3 | Data Preprocessing | 32 |
| 3.4 | Machine Learning models and algorithms | 33 |
| 3.5 | Experimental setup and training | 34 |
| 3.5.1 | Model Development and Experiment Iteration | 35 |
| 3.5.2 | Hyperparameter Settings | 35 |
| 3.6 | Evaluation metrics and validation | 37 |
| 3.7 | Limitations and assumptions | 38 |
| 4 | Results | 39 |
| 4.1 | Overview | 39 |
| 4.2 | Experimental Results | 39 |
| 4.2.1 | Main Pipeline Results | 39 |
| 4.2.2 | Batch Pipeline Results | 40 |
| 4.3 | Comparative Analysis | 44 |
| 4.4 | Summary of Key Findings | 45 |
| 5 | Conclusions | 47 |
| 5.1 | Summary of Research and Findings | 47 |
| 5.2 | Contributions and Significance | 48 |
| 5.3 | Practical Implications and Recommendations | 48 |
| 5.4 | Limitations and Suggestions for Future Research | 49 |

| | |
|--------------------------------------|-----------|
| 5.5 Final Statement | 50 |
| Bibliography | 52 |
| A Appendix | 59 |
| A.1 Classification Reports | 59 |
| A.1.1 Main Pipeline | 59 |
| A.1.2 Batch Pipeline | 61 |

List of Figures

| | | |
|-----|--|----|
| 2.1 | Hot spot heatmap generated via DBSCAN (Kamh et al. 2024). | 7 |
| 2.2 | Shallow model classification. | 9 |
| 2.3 | Structure of a neural network: inputs and weights (Migliaccio and Iannone 2023). | 11 |
| 2.4 | Example of a CNN architecture, used for classification of optical transmission anomalies (Baldini and Cerutti 2023). | 11 |
| 2.5 | Example of a GAN architecture (Popuri and Miller 2023). | 12 |
| 2.6 | The DevOps lifecycle (Reddy et al. 2024). | 13 |
| 2.7 | An example of the MLOps lifecycle, horizontally divided per environment and vertically divided per area of expertise (Testi et al. 2022). | 14 |
| 3.1 | Main pipeline which ingests data, trains a model and evaluates it. | 22 |
| 3.2 | CI pipeline. When the remote branch is updated, a workflow is triggered which creates the updated Job. A self-hosted runner picks it up and executes it in the MLRun namespace in the cluster. | 25 |
| 3.3 | Functions are stored in the MLRun DB and can be found in the MLRun UI, when navigating to the ML Functions tab in a project. | 27 |
| 3.4 | Flowchart of main_pipeline.py. | 28 |
| 3.5 | Flowchart of batch_pipeline.py. | 29 |
| 3.6 | Jobs generated and executed by the pipelines. | 30 |
| 3.7 | Menu showing parameters of main-pipeline run. | 31 |
| 4.1 | Comparison of algorithm performance across accuracy, weighted average F1-score, and macro average F1-score. | 40 |

| | | |
|-----|---------------------------------------|----|
| 4.2 | Accuracy comparison. | 42 |
| 4.3 | Weighted F1-score comparison. | 42 |
| 4.4 | Macro F1-score comparison. | 43 |

List of Tables

| | | |
|------|---|----|
| 3.1 | Hardware Configuration | 34 |
| 3.2 | Software Environment | 35 |
| 3.3 | Model Development and Experiment Iteration Process | 35 |
| 3.4 | Hyperparameter Settings for Each Model | 36 |
| 4.1 | Summary of Macro Average and Weighted Average Metrics Across Algorithms . | 40 |
| 4.2 | Summary of Macro Average and Weighted Average Metrics Across Algorithms, Baseline Evaluation | 41 |
| 4.3 | Summary of Macro Average and Weighted Average Metrics Across Algorithms, Post-Retraining Evaluation | 41 |
| A.1 | Classification Report for Decision Tree Classifier | 59 |
| A.2 | Classification Report for Random Forest Classifier | 59 |
| A.3 | Classification Report for XGBoost Classifier | 60 |
| A.4 | Classification Report for MLP | 60 |
| A.5 | Decision Tree - Evaluation Baseline | 61 |
| A.6 | Decision Tree - Evaluation after retraining | 61 |
| A.7 | Random Forest - Evaluation Baseline | 62 |
| A.8 | Random Forest - Evaluation after retraining | 62 |
| A.9 | XGBoost - Evaluation Baseline | 63 |
| A.10 | XGBoost - Evaluation after retraining | 63 |
| A.11 | MLP - Evaluation Baseline | 64 |
| A.12 | MLP - Evaluation after retraining | 64 |

Listings

| | | |
|-----|---|----|
| 3.1 | Defining a Kubernetes job using MLRun's <code>code_to_function</code> | 23 |
|-----|---|----|

Introduction

1.1 Background

Traffic accidents continue to take a prevalent spot among the leading causes of death for all ages. Despite stellar technological advances in the past decades, the number of annual traffic accidents has only been reduced slightly. Since 2018, the number of annual fatalities caused by traffic accident injuries has declined from 1.35 to 1.19 million¹. While every human life is vitally important and the prevention of such deaths benefits society in a multitude of ways, the price paid for mobility remains too high. Endeavors assisted by Machine Learning (ML) have already been undertaken to introduce traffic accident prediction and prevention to urban areas. However, the challenge lies in creating an agent which is safely and consistently deployed and operated, as well as updated based on new events, in the rapidly changing urban traffic landscape. In this study, the concept of MLOps (Machine Learning Operations) will be explored as a paradigm in order to build a system equipped to tackle this challenge.

1.2 Research problem

The prediction and prevention of traffic accidents rely on the effective analysis of complex data in order to uncover shrouded trends. Traditional ML approaches have produced helpful results, but still fall short in terms of adaptability and near real-time data integration (Mäkinen 2021). This study documents the design, development and deployment of an end-to-end MLOps ecosystem, which achieves reliable traffic accident prediction through continuous model learn-

¹World Health Organization. (2023).

ing and deployment. It will address the identification and analysis of factors which are strongly correlated with traffic accidents, and by implementing an MLOps approach, the developed predictive model will be able to extract and provide actionable insights. Authorities can leverage the provided insights to implement preventative measures, potentially reducing the number of accidents that occur. An emphasis will be placed on making the system as scalable and portable as possible, facilitating its application on numerous pilot areas of interest.

1.3 Aims and objectives

This dissertation aims to achieve the design, development and deployment of an end-to-end MLOps ecosystem, which achieves reliable traffic accident prediction and is continuously trained on new data. The fundamental components which comprise the ecosystem are 1) the ML model, 2) the data ingestion and preprocessing pipelines, 3) the continuous integration, learning and deployment pipelines, and 4) the infrastructure management.

Therefore, the primary objectives are identified as:

1. Locate a dataset which provides data on traffic accidents in the area of study.
2. Design the ecosystem architecture.
3. Evaluate and select methodologies and tools for fundamental components.
4. Train the ML model.
5. Implement the MLOps components to automate the lifecycle.
6. Develop a platform to display data to the end-user.

Additionally, there are three significant secondary objectives:

1. The system must be easily manageable, meaning it should not require specialized knowledge to operate.
2. The system must be reliable, meaning quality standards need to be enforced and met before any new updates.

3. The system must be continuous, ensuring uptime and service availability.

1.4 Research questions

To meet the objectives stated previously, this study seeks to answer specific research questions that address critical aspects of the proposed ecosystem. Given the preexisting research in ML-based traffic accident prediction, relevant questions for this research are:

- *Which are the factors mostly correlated with traffic accidents, and how can one utilize those insights to predict accidents?*
- *What is the spatiotemporal prediction accuracy this system can achieve?*

1.5 Theoretical framework

The introduction of DevOps (Development and Operations) has marked a shift in the way modern teams approach software development, as the demand for rapid releases of high-quality software increases steadily (Karunaratne et al. 2024). This demand increase has been equally reflected among ML projects, giving rise to the concept of MLOps. MLOps represents the application of DevOps principles and practices on ML projects. Unlike traditional software projects, ML initiatives must address additional requirements such as the need to ensure model accuracy, test for model bias, and monitor data and concept drift. MLOps provides a structured approach to automation which can enable engineers to achieve quick, reliable releases and to independently manage their ML workflows. This project will adopt an MLOps approach to its implementation, in order to streamline several development stages so that Continuous Learning (CL) and redeployment of the model can be feasible.

1.6 Research significance

The subject of predicting traffic accidents in an MLOps context is rather new, and thus, very limited exploration of it has been conducted. In IEEE Xplore, searching the query ("traffic acci-

dents" AND "MLOps") returned no results whatsoever. For reference, the combination ("traffic accidents" AND "ML") returned 2246 results. As smart cities become more popular due to the vast amount of real-time data they can provide, software such as this can leverage this crucial opportunity. In a setting where appropriate technology can record traffic accidents in real-time and gather information regarding the weather conditions, number of vehicles involved, and road condition to name a few, a model which learns continuously may prove extremely useful to authorities looking to prevent such events. By enabling the model to automatically learn as new data is ingested, the ecosystem ensures predictions are made with the most recent insights considered. Additionally, by automating parts of the lifecycle, MLOps can reduce the amount of manual operations needed, such as retraining, monitoring and infrastructure management. This may appeal to authorities with tight financial constraints, as the system is designed to be autonomous and not require human supervision at all times.

1.7 Scope and limitations

This dissertation focuses on creating an MLOps solution for traffic accident prediction in the area of study. The scope of this project can be directly extrapolated from the project's aims and objectives as defined in section 1.3. A significant potential constraint is the availability of recent, granular, balanced data regarding traffic accidents in the area of interest. Additionally, all software used in the ecosystem needs to be open-source, and the data used to train the model must be publicly available, as transparency and reproducibility are of high importance.

1.8 Dissertation outline

This section serves as a roadmap for the dissertation, and provides the reader with a clear overview of the document's flow and structure.

1. Introduction: The introduction establishes the context of the study, specifies the research problem and identifies gaps in existing research. It also defines the research questions, aims and objectives, and discusses the significance and scope of the study.

2. Literature Review: The literature review surveys existing studies on the relevant topics, such as MLOps, traffic accident prediction and CT methodologies. By evaluating previous work, it establishes the theoretical foundation for this project.
3. Methodology: The methodology chapter includes the high-level system design adopted to address the aims and objectives. It describes the ecosystem architecture, tools and paradigms selected, as well as the dataset utilized. This chapter also outlines the evaluation process used to measure whether the targets were met.
4. Results and Discussion: In this chapter, the outcomes of the study are presented, including the developed MLOps ecosystem and its evaluation results. It discusses the findings in context of the research questions. Insights are extracted from the evaluation data.
5. Conclusion: The conclusion summarizes the dissertation's findings and discusses their impact. The insights collected in the previous chapter are translated into actionable advice. This chapter discusses the project's next steps and highlights potential research avenues.

Literature Review

2.1 Traffic accidents analysis

Every year, more than 1.1 million lives are cut short due to traffic accidents. Traffic accidents are the leading cause of death for individuals ages 5-29. Additionally, an estimated 20 to 50 million people suffer non-fatal injuries, with many incurring a disability.¹ The socioeconomic turmoil caused by such accidents is significant and includes, but is not limited to, lost output and productivity, medical costs, human suffering, decline in quality of life for victims and their families, property damage, and administrative costs (Bougna et al. 2022). While some progress has been made in reducing the number of accidents, that number still soars high and massively impacts societies worldwide.

2.1.1 Spatiotemporal hotspots

Hotspots can be viewed as areas which have a statistically significant concentration of traffic accidents. Unraveling such patterns is instrumental in taking proactive measures, such as improving road infrastructure, adjusting traffic signals, and launching localized safety campaigns, which are likely to reduce accidents. Traditionally, researchers have relied on methods such as KDE (Kernel Density Estimation) and spatial autocorrelation (Le et al. 2022), statistical clustering with algorithms such as DBSCAN (Density-Based Spatial Clustering of Applications with Noise) (Kamh et al. 2024), and geospatial analysis using GIS tools (Hazaymeh et al. 2022) to identify hotspots.

¹World Health Organization. (2023).

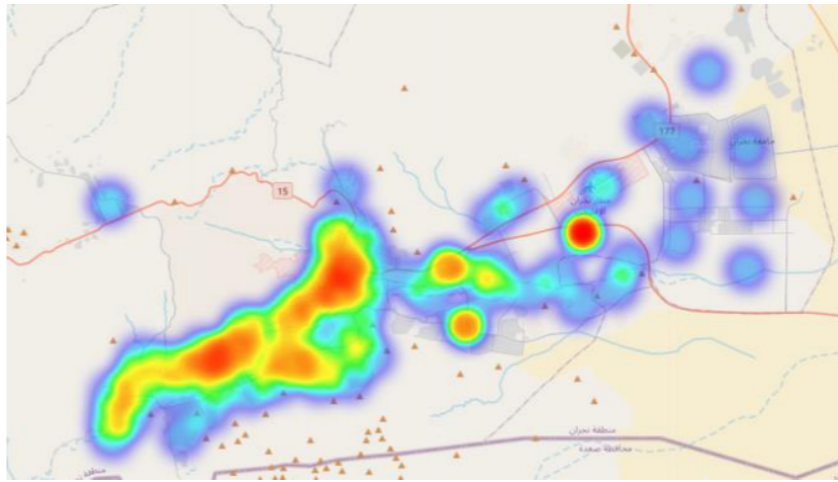


Figure 2.1: Hot spot heatmap generated via DBSCAN (Kamh et al. 2024).

While the reliability of these methods is proven, recent attempts have been made to leverage ML for this purpose. Notably, Karimi et al. 2023 used Self-organizing Maps (SOM) to identify hotspots based on Crash Frequency (CF), the ranking of Equivalent Property Damage Only (EPDO), the Crash Rate (CR), the Empirical Bayes (EB), and the societal risk-based method.

2.1.2 Accident severity classification

Severity classification is the act of labeling an injury or accident based on the harm or damage it has caused. Many systems are used globally in multiple fields, and for multiple purposes. Two interesting examples are the KABCO scale and the Abbreviated Injury Scale (AIS). KABCO is a 5-point scale in which K = Killed, A = Incapacitating injury, B = Severe Injury, C = Minor Injury, O = Only property damage. AIS takes a slightly different approach, as it requires trained medical staff to evaluate the person's injuries and assign a numeric score. A score of over 3 is generally considered a serious injury. In the case of traffic accidents, where non-medical professionals such as police need to quickly respond and determine the accident severity, KABCO is an appropriate choice (Burch et al. 2014). In Great Britain, a more compact scale which includes the values "Fatal", "Serious" and "Slight" is used (Department for Transport 2024). Accurate severity prediction is crucial for providing targeted solutions, which can better inform the distribution of resources such as emergency responders, medical staff, ambulances and traffic police, but also road safety policy decisions in order to prevent future accidents.

2.2 Machine Learning

ML is a branch of Artificial Intelligence (AI) which allows systems to learn from data and make decisions or predictions without being explicitly programmed. ML models are trained on datasets to identify patterns and correlations, which are subsequently used for tasks such as prediction, classification or clustering. The versatility of ML has widened its adoption in industries such as healthcare, finance and technology (Matloff 2024). This chapter will examine the core concepts of ML: data analysis, and various ML algorithms, ranging from common shallow techniques to deep learning.

2.2.1 Data Analysis

In the context of ML, the term data analysis describes the process of handling data before, during and after model training, with the purpose of improving model accuracy and performance. This chapter provides an overview of the most common practices in data analysis.

1. **Data collection:** Data collection is the process of gathering raw data. It can be achieved through searching through databases, spreadsheets, APIs or with the use of web scraping techniques (Quantum 2024).
2. **Data preprocessing:** Data preprocessing is a general term which refers to the steps which transform a raw dataset into one suitable for analysis. It encapsulates processes such as data cleaning, normalization, and standardization (Parashar et al. 2023). Additionally, handling missing data with imputation or deletion techniques, as well as handling outliers, can have significant impact on model training and performance (Galli 2024).
3. **Feature engineering:** Feature engineering refers to the identification, transformation and creation of highly relevant features for the model. Effective feature engineering can greatly improve model performance, interpretability and robustness (Galli 2024).
4. **Data splitting:** Data splitting describes the separation of the dataset into training, testing and validation subsets. This practice can assist in the detection and prevention of

model overfitting, ensuring adaptability to new, unseen data and enhancing the model's generalization capabilities (Quantum 2024).

2.2.2 ML models

ML models can be divided into two major categories: Shallow Learning (SL) and Deep Learning (DL).

Shallow learning (SL)

SL is characterized by computational efficiency and interpretability, and works best with smaller, simpler datasets. It is usually separated in supervised and unsupervised learning. Supervised learning uses labeled data to train models for specific outputs, whereas unsupervised learning detects patterns or structures in unlabeled data (Ankita et al. 2023). Some commonly used shallow algorithms are seen in Figure 2.2.

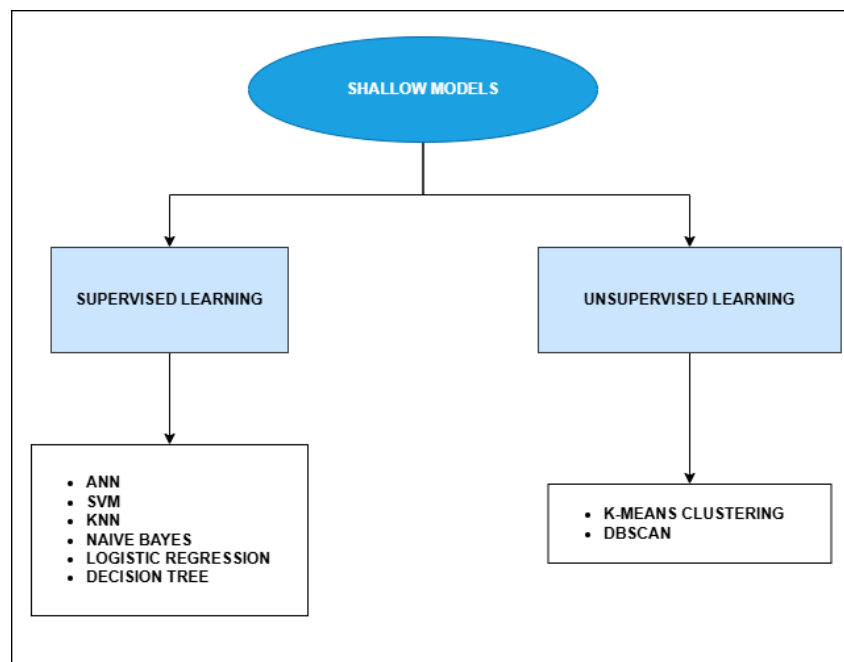


Figure 2.2: *Shallow model classification.*

According to Singh 2022, Logistic Regression (LR) predicts the likelihood of two possible outcomes, making it useful for simple classification problems. Support Vector Machines (SVM) separate data into categories by drawing the best possible dividing line or boundary. k-Nearest Neighbors (KNN) assigns a label to a data point based on the labels of its closest neighbors.

Naive Bayes (NB) uses probabilities to predict categories, assuming that all features are independent of each other. Decision Trees (DT) split data into branches based on specific rules, creating a structure that is easy to understand.

For grouping data, k-Means divides points into a set number of clusters by finding the best group for each point. DBSCAN identifies groups of data points based on how close they are to each other and can handle noisy data well. Artificial Neural Networks (ANNs), with a single hidden layer, mimic how brains process information by passing data through connected layers.

Shallow models include older models proposed before 2006, and they include Artificial Neural Networks (ANNs) with a single hidden layer, unlike the latter Deep Neural Network (DNN) iterations which come with multiple hidden layers (Teixeira et al. 2024).

Deep Learning (DL)

Deep learning is a subfield in ML which deals with the algorithms that, on a much lower level, attempt to mimic the human brain. These methods improve on their shallower counterparts by automatically extracting relevant features from raw data, without the need of the manual intervention of a domain expert. This makes DL models suitable to tackle more general, abstract problems (Nikhil Ketkar 2021). Furthermore, DL is distinguished by its multilayer structure. In traditional feed-forward networks, each neural layer receives their inputs only from the previous one, and conveys its outputs only to the following layer. The appropriate selection of layers and hyperparameters can be considered as the art of deep learning (Glassner 2021).

Similar to traditional ML models, deep learning models can be classified in two categories: discriminative (supervised) and generative (unsupervised). Well known examples for discriminative models include Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), while Generative Adversarial Networks (GANs) and Auto-Encoders (AEs) are eminent examples of generative models (Shiri et al. 2024).

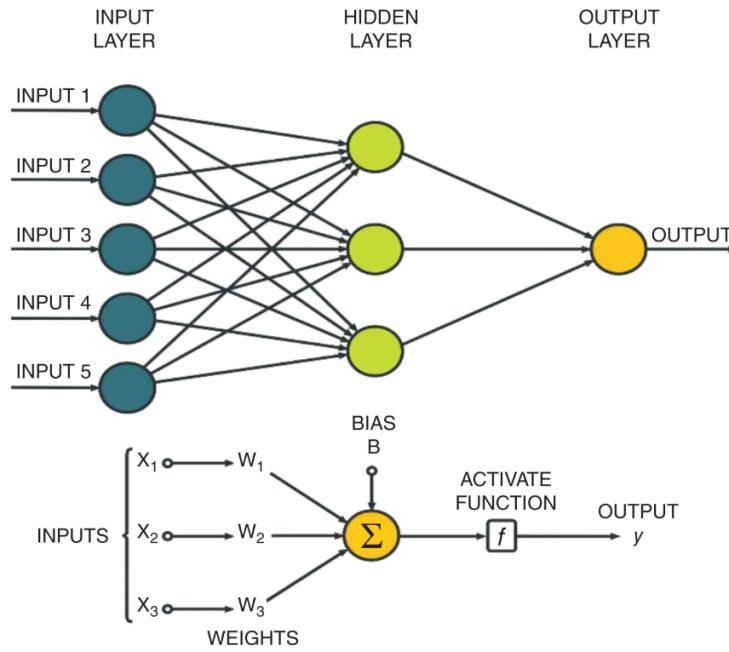


Figure 2.3: Structure of a neural network: inputs and weights (Migliaccio and Iannone 2023).

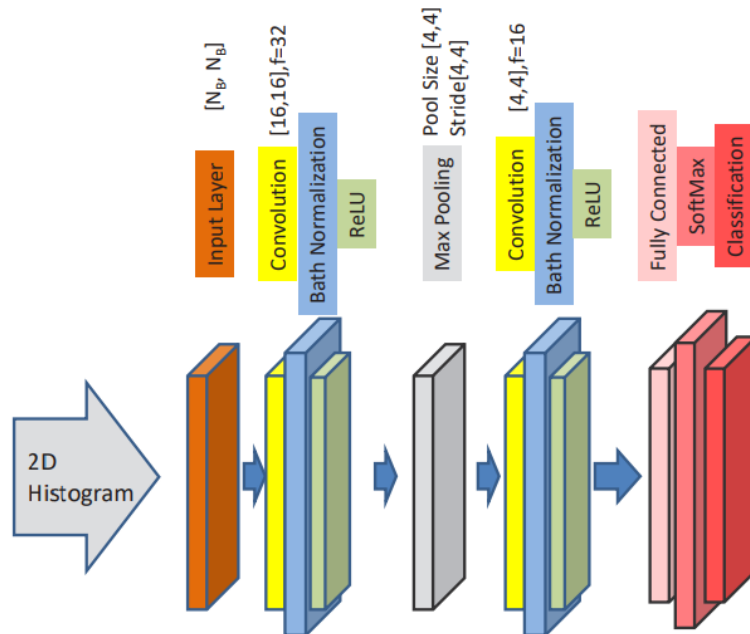


Figure 2.4: Example of a CNN architecture, used for classification of optical transmission anomalies (Baldini and Cerutti 2023).

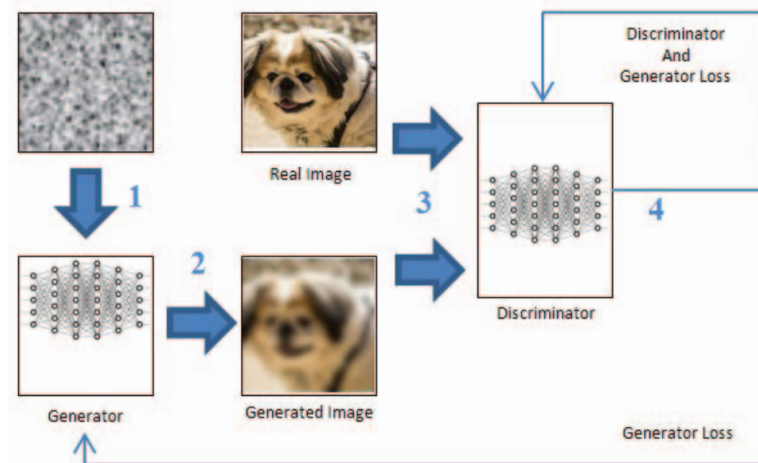


Figure 2.5: Example of a GAN architecture (Popuri and Miller 2023).

2.3 Contemporary Operations

2.3.1 DevOps

DevOps is a conjugation of the words "Development" and "Operations". While there is no single universally accepted definition of the term, it is a common understanding that DevOps refers to a specific mindset and set of principles that an entity applies to the software development life cycle. In the recent past, when software development teams focused on shipping new features as fast as possible, and IT operations teams prioritized maintaining system stability, friction grew between the equally business-critical disciplines. DevOps was introduced as a solution to alleviate this friction, to encourage collaboration and communication between development and operations teams, and facilitate the rapid development and deployment of high quality software (Hattori 2024). DevOps is characterized by an emphasis on automation, which presents itself through the following best practices:

1. **Continuous Integration (CI):** As one of the pillars of software development automation, CI pipelines ensure the latest version of source code adheres to quality standards, and is readily available for release (S. Gupta et al. 2022).
2. **Continuous Delivery/Deployment (CD):** By automatically deploying the latest official version of a software solution to production, engineering teams can deliver service up-

dates without manual release overhead (Reddy et al. 2024). The difference between continuous delivery and deployment is subtle; continuous delivery promises to always have the latest version of software ready for deployment, but requires human instigation to actually trigger the deployment. On the other hand, continuous deployment automatically deploys software to production, eliminating the need for human intervention at all stages.

3. **Microservices:** The microservice architecture is based on the decoupling of services and infrastructure of software components. Each microservice is deployed to an isolate environment and is treated as its own independent product, enhancing development flexibility (Roh et al. 2023). Containerization and orchestration technologies such as Docker and Kubernetes play an integral part in efficient microservice management.
4. **Infrastructure as Code:** This paradigm enforces the automation of infrastructure provisioning and configuration, reducing manual administration. A robust implementation enhances scalability and consistency in system management. Additionally, infrastructure version control enables seamless rollback, in the form of reverting source code commits, since all infrastructure changes are made through Git (S. Gupta et al. 2022).
5. **Monitoring:** Monitoring platforms provide actionable insights to engineering teams and business stakeholders, allowing them to make informed decisions regarding a software system's performance and reliability.

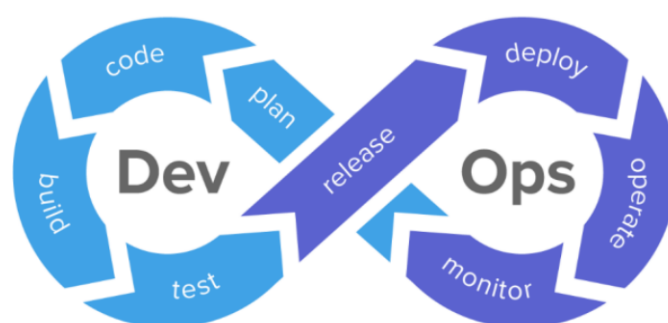


Figure 2.6: *The DevOps lifecycle (Reddy et al. 2024).*

Cross-team communication and collaboration, as well as agile management, were excluded from the list above, even though they are necessary practices for organizations attempting to

adopt a DevOps mindset. This project will focus on the technical details, in order to produce a technically sound, fully functional MLOps ecosystem.

2.3.2 MLOps

The emergence of cloud computing has granted accessibility to powerful computational resources. This has elevated ML into a standardized approach to tackling complex problems. As ML projects rise in popularity, the need to streamline processes remains adamant. MLOps is a set of principles and practices which encompass the ones in DevOps, but are adapted to accommodate the domain-specific requirements of ML. Therefore, one of the primary goals of MLOps is to automate the numerous steps of the ML process, which are illustrated in Figure 2.7.

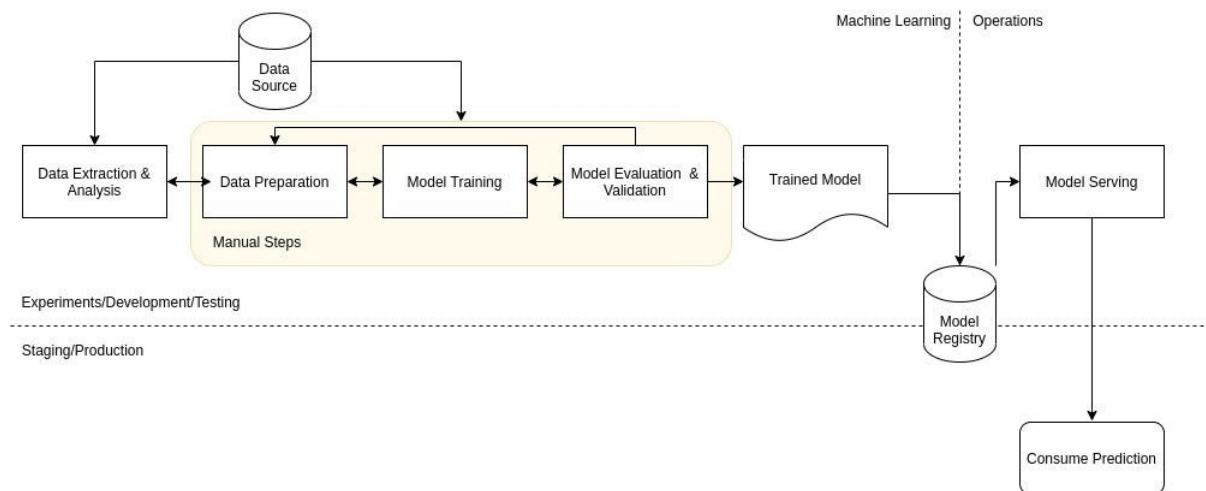


Figure 2.7: An example of the MLOps lifecycle, horizontally divided per environment and vertically divided per area of expertise (Testi et al. 2022).

DevOps utilizes version control for source code and infrastructure. MLOps inherits these use cases as an extension of DevOps, and introduces ML related versioning paradigms on top. The versioning of data, models and pipelines is absolutely essential for MLOps, as processing logic, data sources and the produced models tend to change over time to adapt to emerging changes (Haviv and Gift 2023). MLOps can be considered as the intersection of three major disciplines: DevOps, data engineering and ML (Testi et al. 2022). In the following chapters of this literature review, the focus will be shifted towards the ML and data engineering aspects of MLOps. To be precise, ML will be examined through the lens of Continuous Training (CT) and

Continuous Learning (CL) and data engineering will be approached via data handling, such as storage, versioning and preprocessing automation.

Continuous training and learning

CT is defined as the series of steps a system takes in which a model is retrained with novel data. Naturally, it is usually bundled with model evaluation pipelines as well as deployment to production, even though they are not inherent parts of CT (Kreuzberger et al. 2023). In fact, it can be considered as an intermediary step between CI and CD, as model retraining would occur after adaptations to the source code, but before model deployment. CT allows a model to be retrained on a new dataset, and is usually triggered via a condition; this condition may be a time interval which has lapsed, or a metric threshold which is breached (Garg et al. 2021). CT is conceptually similar to continuous learning (CL), and while their output is seemingly the same—an updated model—their implementation differs significantly. CL enables a model to undergo small, incremental retrainings, where it gradually builds on pre-existing knowledge (Ngo-Ho et al. 2023). This approach is particularly useful when there is a sizeable and constant stream of incoming data, rendering frequent model training rather costly and highly time and resource-consuming. On the contrary, CT always opts to train the model on the totality of the dataset, replacing the previous model with an entirely new, updated and improved version. While at first glance this may seem like an inferior approach, it offers several benefits. Implementing a CT pipeline is much simpler than a CL pipeline, since retraining triggers are clearly defined. This also greatly enhances experiment reproducibility, a key principle of MLOps. CT provides better control over the model updates, as there is a clear validation phase before deployment, as well as mitigates risks by being able to easily track and revert poor model updates, thus increasing model stability. Additionally, CT can work surprisingly well with large-volume datasets, given its batch processing capabilities. This method of data handling is ideal for datasets which are updated at regular intervals or with predictable changes.

Data versioning in MLOps

The importance of data cannot be overstated, since ML models are derived from data. There has been a recent industry shift to data-centric AI, meaning that the importance of data in building ML and DL models is elevated significantly (Liu 2022). In MLOps, data versioning is considered an essential practice which not only drives product development, but enhances data quality, model reproducibility, data auditability and compliance, as well as collaboration and dependency tracking (Haviv and Gift 2023). Some open-source data versioning tools are DVC, MLFlow, MLRun, GuildAI and Decibel (Pacheco et al. 2024; Haviv and Gift 2023; Zarate et al. 2022).

Data pipelines in MLOps

A data pipeline is a series of processes which automate movement and transformation of data from one system to another. They facilitate the efficient collection, processing and delivery of data. Two common types of data pipelines are:

ETL Pipelines (Extract, Transform, Load):

- **Extract:** Data is collected from various sources, such as databases, APIs, or files.
- **Transform:** Data is cleaned and altered until it reaches a digestible form for the model.
- **Load:** Load describes the final step of loading the transformed data into the model, or a data storage system.

Notably, ETL pipelines typically occur after data ingestion, which is the process during which the dataset is updated based on various sources such as databases, APIs or files (Espenchutz 2023).

Batch Processing Pipelines:

- Batch processing involves collecting data over a period of time and processing it in bulk.
- It is ideal for tasks that don't require real-time processing, such as generating daily reports or aggregating data for historical analysis.

- Batch pipelines often work with scheduled jobs that trigger data processing at specific intervals, or based on specific metric thresholds.

Batch processing is efficient for handling large datasets and is commonly employed when real-time insights are not necessary (Haviv and Gift 2023).

2.4 ML in traffic accident prediction

This section thoroughly examines related work in utilizing ML to predict traffic accidents. The MLOps system which is the subject of this dissertation mainly serves one purpose: to automate various tasks and steps in order to accelerate and improve the development and deployment of the ML models. The core of the system is the collection of models which are making the predictions. A system which predicts traffic accidents needs to provide three crucial pieces of information: the time, location, and severity of the accident. Given the multifaceted requirements of the problem, three separate ML models must be implemented. The output of one model can be provided to the next model as input, to introduce further constraints and increase prediction specificity (Santos et al. 2021).

The LSTM (Long short-term memory) algorithm is a type of deep neural network based on RNNs. Zhang et al. 2020 successfully used LSTM in order to identify crime hotspots in a research area in China using historical crime data. A comparison of LSTM with KNN, Random Forest (RF), SVM, NB classifier, and CNN, showed that the prediction accuracy of LSTM model were superior to the competing algorithms. Additionally, adding environmental covariates further improved the accuracy of the model.

Alhaek et al. 2024 used a combination of a CNN and a BiLSTM (Bi-directional LSTM) to extract spatiotemporal information from the dataset, and perform severity classification on accidents that happened in London and Liverpool in the UK. Their method consistently outperformed SVM, XGBoost, 1D CNN, DNN and CNN-LSTM alternatives, achieving an F1-score of 0.88 for London and 0.85 for Liverpool.

U. Gupta et al. 2022 applied KDE to identify hotspots from a dataset which contained over

1.4 million accidents in the UK for the years 2005-2018. They also used a range of shallow models for the severity classification. A combination of RF and SMOTE (Synthetic Minority Oversampling Technique) yielded the highest F1-score of 0.93. However, it is noteworthy that among the best performing models, XGBoost and RF, the use of SMOTE did not affect the accuracy or F1-score of XGBoost, and only increased the accuracy when used with RF, while still not affecting the F1-score.

Islam et al. 2022 compared RF, LR and XGBoost in severity classification for traffic accidents in an area in Al-Ahsa, Saudi Arabia. RF performed marginally better than XGBoost and significantly better than LR, with an F1-score of 0.85 for fatal accidents and 0.96 for accidents causing serious injury. GIS-based spatial autocorrelation was used for hotspots identification. Additionally, the feature importance score from RF suggests that faulty tires, not giving way, sudden turning and running over were the most critical causes for severe accidents.

Santos et al. 2021 attempted to develop a predictive model for future road accidents by using a variety of supervised and unsupervised learning models, such as DT, RF, LR, NB, DBSCAN and hierarchical clustering. A rule-based model using the C5.0 algorithm performed best in identifying the most relevant severity factors. Negative sampling was used in order to maximize classification accuracy. Additionally, the research suggests that the RF model could prove useful for forecasting accident hotspots.

Bao et al. 2024 proposed a Back Propagation Neural Network (BP), also known as a Multilayer Perceptron (MLP), with a single hidden layer is suitable for classifying and predicting the traffic accidents on national and provincial highways. This work showcases an interesting case in model efficiency: two hidden layers would take over 20 times longer to train, and would result in a 2.95% increase in test set accuracy, and a 0.7% decrease in training set accuracy. This result accentuates the potential of diminishing returns when deciding on an ML approach.

2.5 Research Gaps

While ML has been used extensively to produce meaningful results in traffic accident prediction and classification, MLOps has not been explored as a potential improvement in this

domain. In IEEEXplore, a search with the query ("traffic accidents" AND "MLOps") yielded no matching results. In this dissertation, MLOps will be examined as a complimentary set of practices to apply on top of existing methodologies to improve multiple aspects of model development, performance and deployment.

Methodology

3.1 Introduction

Building upon the identified research gaps and objectives in the relevant literature, and particularly the underexplored potential of MLOps in traffic accident severity classification, this methodology outlines the systematic approach undertaken to address these gaps. By adopting MLOps practices, this study aims to improve multiple aspects of model development, performance and deployment, as well as enhance automation, flexibility, scalability and experiment reproducibility, as per section 2.3.2.

3.2 System Architecture

This research leverages MLOps practices to build a continuous learning pipeline for traffic accident severity classification using MLRun, an open-source MLOps framework¹. The original concept was to automate the retraining and deployment of machine learning models when new batches of data were available. The goal was to ensure that models were kept up-to-date with the latest traffic patterns, ultimately improving prediction accuracy and resource allocation.

¹<https://www.mlrun.org/>

3.2.1 Infrastructure Design

MLRun consists of a comprehensive tool suite, providing rich functionality without requiring additional configuration from the user. The MLRun Community Edition¹ consists of:

1. **MLRun**: The MLRun API, MLRun UI and MLRun DB.
2. **Nuclio**: An engine for running serverless functions.
3. **Jupyter**: A Jupyter Notebook UI, integrated with MLRun.
4. **MPI Operator**: A Kubeflow Operator designed to enable distributed training on Kubernetes.
5. **MinIO**: High performance object storage, whose API is compatible with Amazon S3 cloud storage service.
6. **Kubeflow Pipelines**: Reusable end-to-end ML workflows built using the Kubeflow Pipelines SDK.
7. **Spark Operator**: A Kubernetes Operator designed to facilitate specifying and running Spark applications on Kubernetes, using declarative files.
8. **Prometheus Stack**: A cluster monitoring tool, collecting metrics from Kubernetes components. Users can visualize the collected information in the Grafana UI.

To enable the automatic installation and orchestration of the tool suite, MLRun requires to be installed on a Kubernetes cluster via a Helm chart. Kubernetes is an open-source software that primarily excels in orchestrating the deployment, scaling and management of container-based workloads (Agrawal 2023). Three basic concepts that are essential to this project are clusters, pods and jobs. A Kubernetes cluster is a collection of nodes that work together to run, manage, and scale containerized applications. It provides a unified system where resources are pooled and orchestrated to ensure high availability and efficient utilization (Boorshtein et al. 2024). A pod is the smallest and most basic deployable unit in Kubernetes. It encapsulates one or more tightly coupled containers that share the same network namespace and storage

¹<https://docs.mlrun.org/en/stable/install/kubernetes.html>

volumes. Pods are designed to run a single application or closely related processes that must operate together. Jobs are short-lived pods that run until success or failure. A job is better suited for finite workloads, and should not be used as a deployment method for long-running processes, such as web services (Sayfan 2023).

A local Kubernetes cluster provided by Docker Desktop¹ was used to implement this project. Other noteworthy options for running local Kubernetes clusters were Minikube² and k3d³.

A high-level design of the system architecture can be viewed in figure 3.1. This architecture utilises the vendor-agnostic components that MLRun provides, rendering the solution easily portable to various cloud providers.

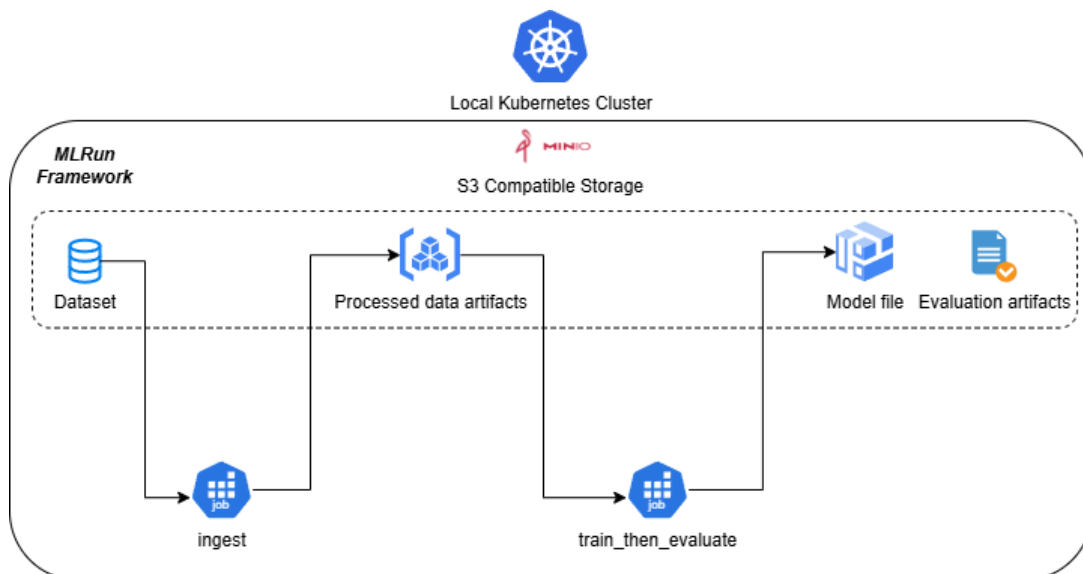


Figure 3.1: Main pipeline which ingests data, trains a model and evaluates it.

Specifically, MinIO serves as the S3 compatible storage service, and any piece of code always runs in a Kubernetes job, resembling serverless functionality. This aligns with the core principles of cloud-native design, which encourages constructing independent applications that do not rely on traditional infrastructure but ensure agility and scalability, enabling seamless transitions to the cloud deployment model (Ahmed 2024).

¹<https://docs.docker.com/desktop/features/kubernetes/>

²<https://minikube.sigs.k8s.io/docs/>

³<https://k3d.io/stable/>

3.2.2 Source code to Kubernetes job

Serverless-like code execution is achieved through a built-in function of the MLRun framework, called `mlrun.code_to_function`¹. This function offers a convenient way to insert code into multiple types of runtime objects, including but not limited to Kubernetes jobs, Nuclio serverless functions, and Spark jobs. The job runtime object was preferred for its simplicity, allowing the python files to be compiled into jobs, creating the illusion of serverless functionality.

Listing 3.1: *Defining a Kubernetes job using MLRun's `code_to_function`*

```
# In update_functions.py:
import mlrun

# ... additional logic

ingest = mlrun.code_to_function(
    name="ingest",
    kind="job",
    filename="ingest.py",
    project=os.getenv('PROJECT_NAME'),
    image=os.getenv('JOB_IMAGE'),
)
```

The `mlrun.code_to_function` method accepts several parameters, each serving a specific role in defining the function's behavior. The key parameters used in this snippet are explained below:

- **name:** Specifies the name of the function being created. In this case, "ingest" is the name assigned to the function.
- **kind:** Defines the type of runtime object. Here, "job" indicates that the function will be executed as a Kubernetes job.
- **filename:** Refers to the Python file (`ingest.py`) containing the function's code. This file will be executed when the job runs.
- **project:** This parameter is set to the value retrieved from the `PROJECT_NAME` environment variable, indicating the project under which the function is registered.

¹https://docs.mlrun.org/en/stable/api/mlrun.html#mlrun.code_to_function

- **image:** Specifies the Docker image to be used for the function. It defaults to an `mlrun/mlrun` image, but in this case provides a custom image with additional dependencies, created from `images/Dockerfile.job` and pushed to DockerHub.

Furthermore, the function `os.getenv` is utilized to retrieve environment variables. This approach is necessary as all project configuration is defined within the `.env/main_config.env` file. Centralizing all experiment parameters in a single configuration file enhances the efficiency of experimentation while maintaining a single source of truth throughout the project.

Special attention should be given to the `image` parameter, which defines the base image of the job created to run the function. This parameter is crucial for the function's execution environment. According to MLRun's documentation¹, the default job image is `mlrun/mlrun`². This image includes a range of popular data science and machine learning dependencies, such as Python, Pip, Pandas, Sklearn, and more. However, if a Python module attempts to import a dependency not included in the default `mlrun/mlrun` image, it will result in an error.

Such was the case in this project when attempting to use the XGBoost algorithm, which is not included in the default image. To resolve this issue, a custom image was created. This custom image is based on the `mlrun/mlrun` image but is enriched with additional dependencies required by the project. The custom image is defined in the `images/Dockerfile.job` file.

The process for updating the job image involves building the custom image from the Dockerfile and pushing it to the user's personal Docker registry. Once the image is uploaded, it is referenced in the `image` parameter of the `code_to_function` method, enabling MLRun to locate the image in the user's personal registry. Since installing MLRun requires Docker registry credentials, MLRun is able to locate the image and use it during job execution.

3.2.3 Continuous Integration

CI is a practice where developers frequently merge code changes into a central repository, followed by automated builds and tests. This ensures early detection of integration issues

¹<https://docs.mlrun.org/en/stable/runtimes/images.html#mlrun-runtime-images>

²<https://hub.docker.com/r/mlrun/mlrun>

and helps maintain high software quality. In this project, CI is an integral part of the MLOps pipeline, facilitating the seamless integration of code changes in the models, data preprocessing scripts, and deployment configurations.

Since the project source code is stored in a GitHub repository¹, a well-rounded, native CI option is GitHub Actions. GitHub Actions is an automation tool enabling professionals to build solutions across the entire GitHub ecosystem, which includes applications for task planning and tracking, client applications, security scans, developer productivity, and CI/CD among others (Kaufmann 2024).

Figure 3.2 outlines the process of leveraging GitHub Actions workflows to update the functions stored in the MLRun DB to include the latest code modifications.

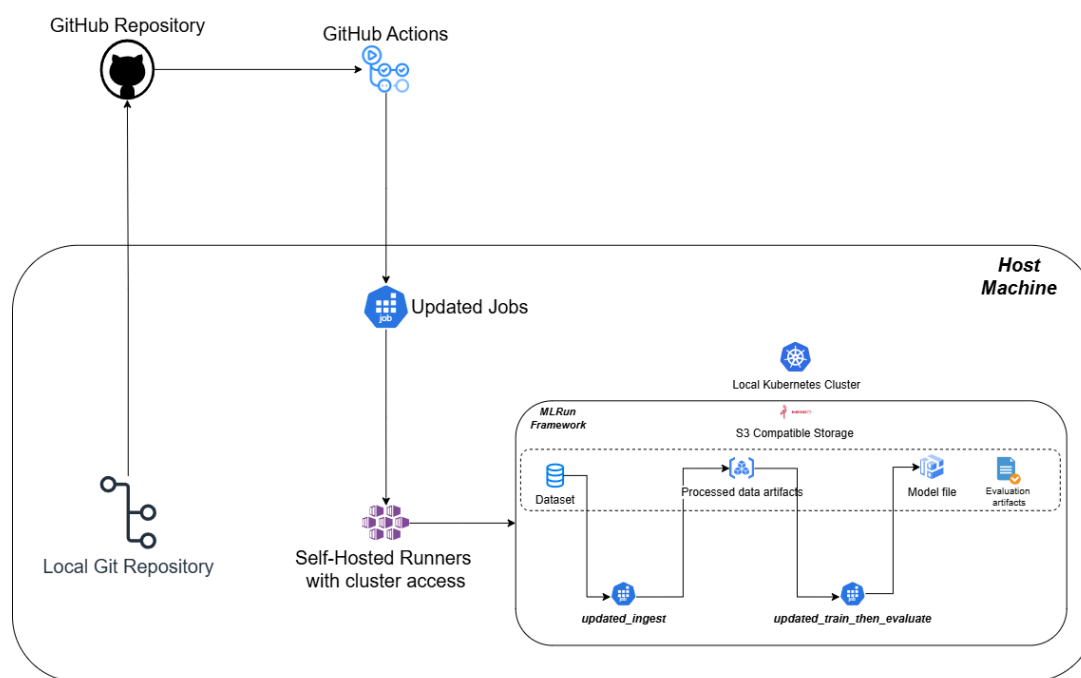


Figure 3.2: *CI pipeline. When the remote branch is updated, a workflow is triggered which creates the updated Job. A self-hosted runner picks it up and executes it in the MLRun namespace in the cluster.*

Self-hosted GitHub Runners

In GitHub Actions, a "runner" refers to a machine or system that executes a GitHub Actions job (Laster 2023). There are two types of runners: GitHub-hosted and self-hosted. GitHub

¹<https://github.com/samismos/traffic-accidents-mlops>

provides free runners for public repositories, offering up to 2000 minutes per month.

However, using GitHub-hosted runners introduces challenges. Since these runners are external to the local network hosting the Kubernetes cluster, connecting to the cluster requires exposing core cluster endpoints—such as the MLRun API, MLRun DB, and MinIO API—via port forwarding. This approach poses several security risks and may also face networking limitations imposed by the local internet provider's rules.

In contrast, self-hosted runners address these concerns, as they do not require incoming traffic from GitHub. Instead, they only need outbound access, which is authorized via the user's GitHub token. Once configured, the self-hosted runner connects to GitHub with an HTTPS long poll to receive job assignments. When a job is created from the workflow file under `.github/workflows`, the runner picks it up and executes it¹. This approach eliminates the need for external access to the local network. Since the self-hosted runner resides within the same network as the Kubernetes cluster, it can refer to the cluster as `localhost`, facilitating seamless interaction with every essential MLRun component.

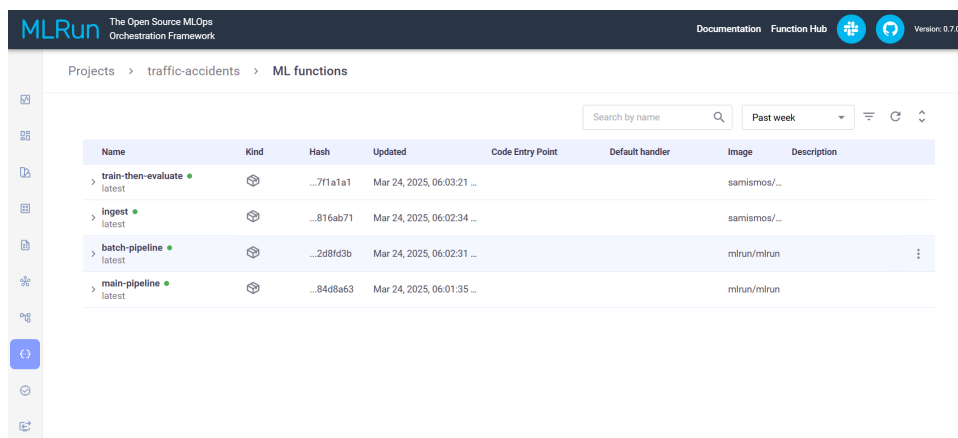
Updating MLRun Functions

This CI pipeline executes two Python scripts: `update_functions.py` and `update_pipelines.py`. These scripts utilize the `code_to_function` method, as discussed in 3.2.2, to update important functions in the project.

The `update_functions.py` updates the code in the individual jobs, namely `ingest.py` and `train_then_evaluate.py`. Meanwhile, `update_pipelines.py` handles the pipelines, which orchestrate the execution of these jobs in a specific sequence, with defined parameters.

The pipelines implement the two primary functionalities required by the project: initial ingestion and model training, as well as batch ingestion and retraining. The pipelines are defined under the `pipelines` directory, as `main_pipeline.py` and `batch_pipeline.py` respectively. ML functions as displayed in the UI can be seen in figure 3.3.

¹<https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners#communication-between-self-hosted-runners-and-github>



| Name | Kind | Hash | Updated | Code Entry Point | Default handler | Image | Description |
|---------------------------------|------|------------|----------------------------|------------------|-----------------|--------------|-------------|
| > train-then-evaluate latest | 📦 | ...7f1a1a1 | Mar 24, 2025, 06:03:21 ... | | | samismos/... | |
| > ingest latest | 📦 | ...816ab71 | Mar 24, 2025, 06:02:34 ... | | | samismos/... | |
| > batch-pipeline latest | 📦 | ...2d8fd3b | Mar 24, 2025, 06:02:31 ... | | | mlrun/mlrun | |
| > main-pipeline latest | 📦 | ...84d8a63 | Mar 24, 2025, 06:01:35 ... | | | mlrun/mlrun | |

Figure 3.3: Functions are stored in the MLRun DB and can be found in the MLRun UI, when navigating to the ML Functions tab in a project.

Main Pipeline

The main pipeline follows a linear flow:

- **Ingest full dataset:** Performs preprocessing on the entire dataset, as defined in the `ingest` job.
- **Train and evaluate model:** Trains the model, evaluates it, and stores the model file and evaluation artifacts.

The main pipeline is excellent for creating initial models and clearly evaluating their performance on the dataset.

Batch Pipeline

The **batch pipeline** involves some additional steps:

- **Ingest batch data:** Performs preprocessing only on the new batch, and stores the processed batch artifacts.
- **Model evaluation reference:** Fetches the latest existing model from the model store, and evaluates it against the whole dataset, in order to have a base reference for model performance.
- **Evaluate on batch:** Fetches the latest existing model from the model store, and evaluates it against the batch data. This shows the model's ability to predict on unknown

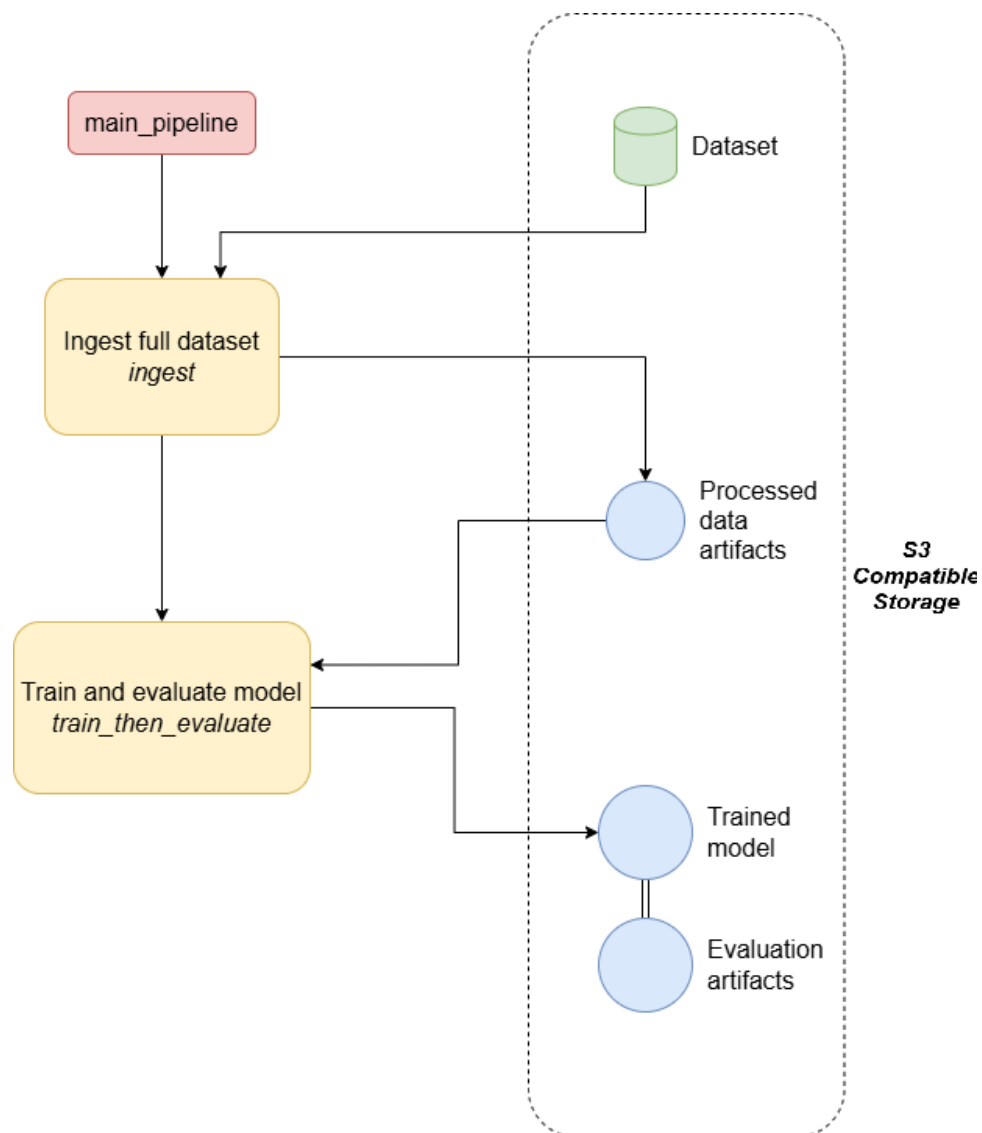


Figure 3.4: Flowchart of `main_pipeline.py`.

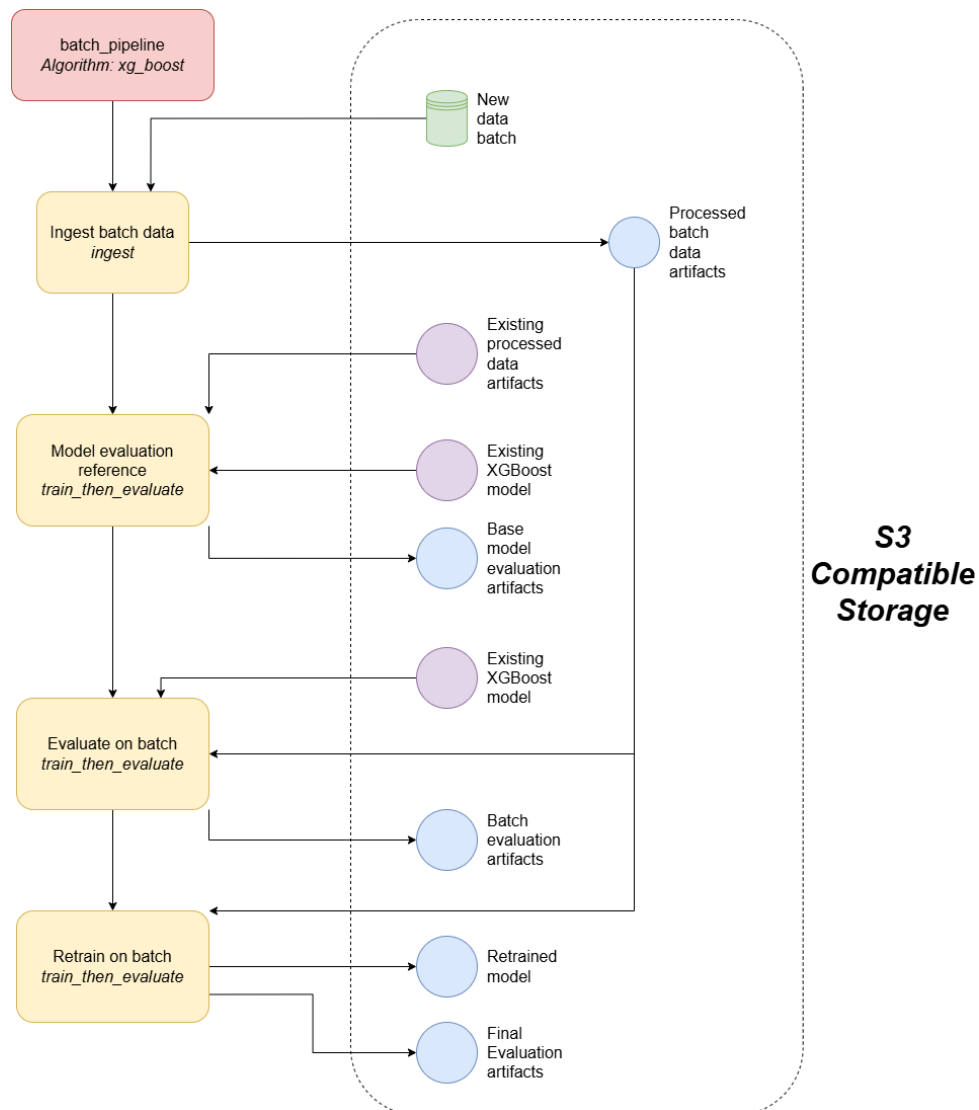


Figure 3.5: Flowchart of `batch_pipeline.py`.

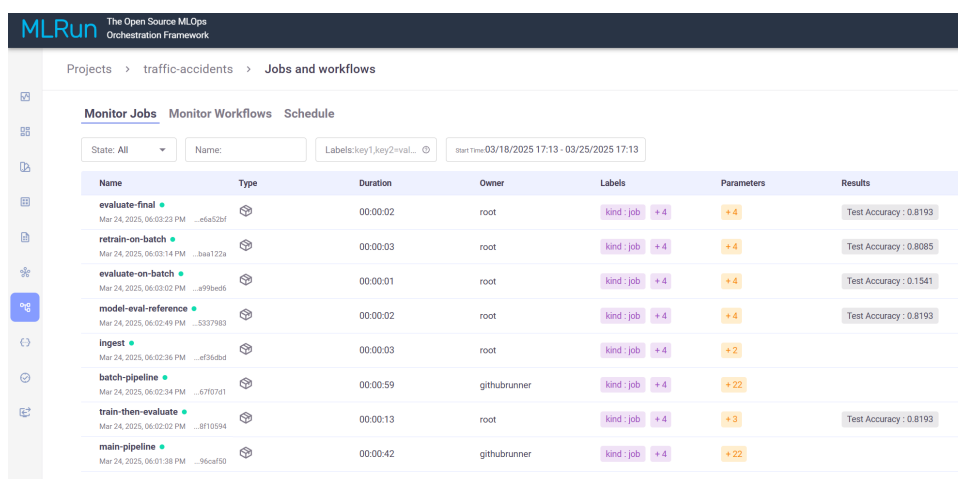
data.

- **Retrain on batch:** Fetches the latest existing model from the model store, fits it with the batch data, and evaluates it. This step is designed to showcase if the model improves its prediction ability from training on new data.

It is important to note that the batch pipeline assumes the existence of models and processed data artifacts. This requires the main pipeline to have been executed at least once before the batch pipeline. This is rather sensible; in any scenario, the initial dataset would have to be ingested and a baseline model should be trained, before any retraining on a batch occurs.

Executing MLRun Functions

Once functions are registered in the MLRun DB, they can be executed without the need to supply the source file every time. This is extremely useful because it allows the user to execute any pipeline directly from the UI. The MLRun UI provides a menu which allows tweaking of execution parameters such as the algorithm, without interacting with the source code. This can be done through the "Jobs and workflows" menu in MLRun UI, as shown in figure 3.6.



The screenshot shows the MLRun UI interface. The top navigation bar includes 'Projects' and 'traffic-accidents'. The main section is titled 'Jobs and workflows' and contains a 'Monitor Jobs' tab. Below the tab are filters for 'State: All', 'Name:', 'Labels: key1, key2=val...', and 'Start Time: 03/18/2025 17:13 - 03/25/2025 17:13'. A table lists several jobs with columns for Name, Type, Duration, Owner, Labels, Parameters, and Results. The jobs include 'evaluate-final', 'retrain-on-batch', 'evaluate-on-batch', 'model-eval-reference', 'ingest', 'batch-pipeline', 'train-then-evaluate', and 'main-pipeline'.

| Name | Type | Duration | Owner | Labels | Parameters | Results |
|--|------|----------|--------------|--------------|------------|------------------------|
| evaluate-final Mar 24, 2025, 06:09:23 PM ...ef552bf | | 00:00:02 | root | kind: job +4 | +4 | Test Accuracy : 0.8193 |
| retrain-on-batch Mar 24, 2025, 06:03:14 PM ...baa1722a | | 00:00:03 | root | kind: job +4 | +4 | Test Accuracy : 0.8085 |
| evaluate-on-batch Mar 24, 2025, 06:03:02 PM ...a99bed6 | | 00:00:01 | root | kind: job +4 | +4 | Test Accuracy : 0.1541 |
| model-eval-reference Mar 24, 2025, 06:02:49 PM ...5337983 | | 00:00:02 | root | kind: job +4 | +4 | Test Accuracy : 0.8193 |
| ingest Mar 24, 2025, 06:02:36 PM ...ef36dbd | | 00:00:03 | root | kind: job +4 | +2 | |
| batch-pipeline Mar 24, 2025, 06:02:34 PM ...670761 | | 00:00:59 | githubrunner | kind: job +4 | +22 | |
| train-then-evaluate Mar 24, 2025, 06:02:02 PM ...8110594 | | 00:00:13 | root | kind: job +4 | +3 | Test Accuracy : 0.8193 |
| main-pipeline Mar 24, 2025, 06:01:38 PM ...96caf50 | | 00:00:42 | githubrunner | kind: job +4 | +22 | |

Figure 3.6: Jobs generated and executed by the pipelines.

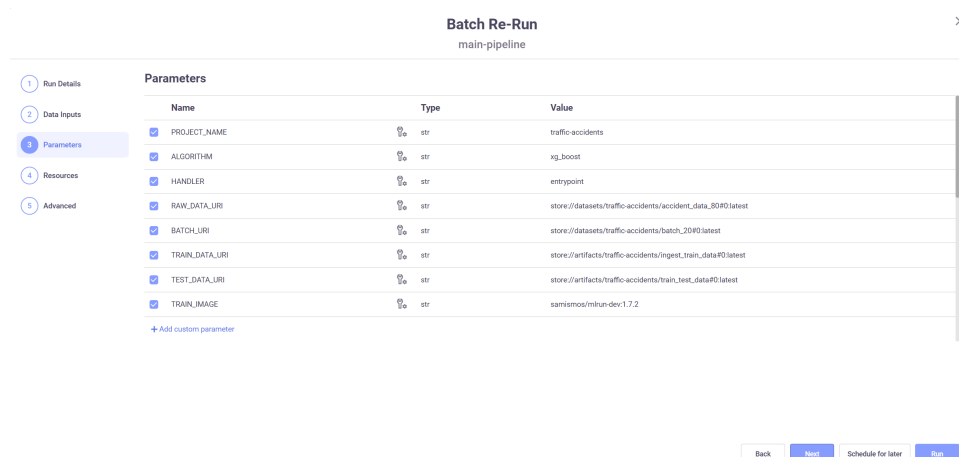


Figure 3.7: Menu showing parameters of main-pipeline run.

3.3 Data collection and preprocessing

The dataset used for traffic accident severity classification is publicly available at Kaggle¹, ensuring transparency and reproducibility. The data ingestion and preprocessing pipeline is implemented using the `ingest.py` script, which leverages a combination of data science libraries including pandas, numpy, and sklearn. This section details the steps followed in the data collection and preprocessing stages.

3.3.1 Data Loading

The data ingestion process begins by loading the dataset through a specified `DATASET_URI` using the MLRun framework. The URI is a reference to MLRun's data store, which serves as an intermediary between the code and the S3 compatible storage. This approach ensures flexibility and integration with the broader MLOps pipeline.

3.3.2 Initial Data Exploration

Upon loading the dataset, the script performs an initial exploratory analysis by:

- Printing the dataset's shape and metadata.
- Checking for missing values.

¹<https://www.kaggle.com/datasets/charliescott556/uk-vehicle-accident-database-2019-2022>

- Generating descriptive statistics.

This step provides an overview of the dataset's structure and highlights any immediate data quality concerns.

3.3.3 Data Preprocessing

Data preprocessing is crucial for ensuring the quality and reliability of the model. The script applies the following steps:

Handling Missing Values

- **Numerical Features:** Missing values are filled with the median to mitigate the impact of outliers.
- **Categorical Features:** Missing values are filled with the mode to preserve the most frequent category.

Label Encoding

- Categorical variables are converted into numerical format using `LabelEncoder` from `sklearn`.

Feature Scaling

- Numerical features, excluding the target variable, are standardized using `StandardScaler` to ensure all features contribute equally to the model's performance.

Data Splitting

- The dataset is split into training and testing sets with an 80-20 ratio.
- Stratified sampling is employed to maintain the class distribution in both sets, preventing model bias towards the majority class.

Handling Class Imbalance

Class imbalance is a common issue in severity classification. To address this, the Synthetic Minority Over-sampling Technique (SMOTE) is applied to the training set:

- Balances the minority and majority classes by generating synthetic samples.
- Ensures the model is not biased towards the prevalent class.

Data Saving and Versioning

Processed datasets are saved as CSV files for consistency and traceability. These files include:

- `X_train_over.csv` and `y_train_over.csv` for the oversampled training set.
- `X_test.csv` and `y_test.csv` for the testing set.

The script utilizes MLRun's artifact logging capabilities for version control, embedding meta-data and maintaining a record of data processing versions.

3.4 Machine Learning models and algorithms

The study implements multiple machine learning algorithms to evaluate and compare performance on the severity classification task. The following models are employed:

- **Decision Tree Classifier:**
 - Simple and interpretable model based on hierarchical decisions.
 - Suitable for datasets with non-linear relationships.
- **Random Forest Classifier:**
 - Ensemble of decision trees with random feature selection.
 - Reduces overfitting by averaging multiple trees.
- **XGBoost (Extreme Gradient Boosting):**

- Gradient boosting framework optimized for performance and speed.
 - Parameters include 100 estimators, max depth of 5, and a learning rate of 0.1.
 - Utilizes the multi-class softmax objective for severity classification.
- **Multi-Layer Perceptron (MLP) Classifier:**
 - Neural network with two hidden layers (100 and 50 neurons).
 - ReLU activation function and adaptive learning rate.
 - Optimized with the Adam solver.

Training and Evaluation

Each model undergoes training and evaluation on the processed dataset:

- Data is split into 80% training and 20% testing sets, with stratified sampling to maintain class distribution.
- Performance metrics include accuracy, classification report, and confusion matrix.
- Results are logged using MLRun, ensuring reproducibility and versioning.
- The confusion matrix and classification report are saved as CSV files for further analysis.

3.5 Experimental setup and training

The experimental setup outlines the environment, resources and procedures used to train and evaluate the machine learning models and operations.

| Component | Specification |
|-----------|--------------------------------------|
| Processor | AMD Ryzen 5700X, 8 core / 16 threads |
| Memory | 32 GB DDR4 |
| Storage | 500GB NVMe SSD |
| GPU | Not available |

Table 3.1: Hardware Configuration

| Component | Version / Specification |
|------------------------|---|
| Host Operating System | Windows 11 Home, Version 24H2 |
| Project Infrastructure | Docker Desktop 4.30.0, Kubernetes v1.29.2 |
| Framework | MLRun 0.7.0 |
| Programming Language | Python 3.9 |
| Version Control | Git 2.45.2.windows.1 |
| ML Libraries | Scikit-learn, XGBoost |

Table 3.2: *Software Environment*

3.5.1 Model Development and Experiment Iteration

| Step | Description |
|--|--|
| Experiment Initialization Methods | |
| Temporary Job Container | Mount and run scripts inside a container built from the job image. |
| MLRun UI | Directly run pipelines from the MLRun UI. |
| CI | Trigger the CI pipeline action in GitHub Actions. |
| Training and Evaluation Process | |
| Data Input | Previously split data is passed into the function as arguments. |
| Model Initialization | Four classification models are initialized: <ul style="list-style-type: none"> • Decision Tree Classifier • Random Forest Classifier • XGBoost Classifier • Multi-Layer Perceptron (MLP) |
| Model Selection | The model corresponding to the "algorithm" parameter is selected. |
| Pretrained Model Check | If "model_uri" is populated, the function fetches the existing model; otherwise, a new model is trained. |
| Training and Evaluation | The selected model is trained, and evaluation metrics are produced. |

Table 3.3: *Model Development and Experiment Iteration Process*

3.5.2 Hyperparameter Settings

Each model was configured with the following hyperparameters:

| Model | Hyperparameters |
|--------------------------|--|
| Decision Tree Classifier | Default parameters |
| Random Forest Classifier | Default parameters |
| XGBoost Classifier | <ul style="list-style-type: none"> • n_estimators: 100 • max_depth: 5 • learning_rate: 0.1 • objective: 'multi:softmax' • eval_metric: 'mlogloss' |
| MLP | <ul style="list-style-type: none"> • hidden_layer_sizes: (100, 50) • activation: 'relu' • solver: 'adam' • alpha: 0.0001 • learning_rate: 'adaptive' • max_iter: 200 |

Table 3.4: *Hyperparameter Settings for Each Model*

3.6 Evaluation metrics and validation

Several important metrics are retrieved in order to evaluate how well the classification models perform when used for the traffic accident severity classification task:

- **Accuracy:** By figuring out the percentage of properly predicted cases out of all instances, this statistic assesses the model's overall accuracy. Despite being a simple statistic, accuracy can be deceptive when there is a class imbalance.
- **Classification Report:** For each class (Fatal, Serious, Slight), the classification report offers a thorough analysis of precision, recall, and F1-score. Every statistic has a distinct function:
 - **Precision::** Shows the percentage of actual positive forecasts among all expected positive forecasts. The model's high accuracy indicates that it produces few false-positive results.
 - **Sensitivity (Recall)::** Indicates how well the model can identify real positive cases. A high recall means that few real positive examples are missed by the model.
 - **F1-Score::** A balanced metric that is helpful in addressing class imbalance, it is the harmonic mean of accuracy and recall.
- **Confusion Matrix:** This is a table used to describe the performance of a classification model by showing the number of correct and incorrect predictions for each class. It provides insight into the types of errors the model makes, allowing for targeted improvements.

To replicate real-world performance, the assessment procedure is carried out on a different test set. Overfitting is reduced and the model's capacity for generalization is examined by dividing the dataset into training and test subsets.

The following metrics are preserved as artifacts:

- **Confusion Matrix CSV:** Assists in visualizing the prediction distribution.

- **Classification Report CSV:** Provides comprehensive performance metrics by class.

Together, these measurements provide a thorough grasp of the models' performance, assisting in the selection of models and subsequent fine-tuning.

3.7 Limitations and assumptions

A number of restrictions might affect the models' functionality and applicability. Class imbalance is the main problem, with the minority classes (Fatal and Serious) making up a lower percentage of the dataset than the Slight class. Biased models that perform well on the majority class but poorly on important minority classes may result from this imbalance.

Lack of GPU resources is another major drawback that prevents the training of more intricate, computationally demanding models like deep neural networks or bigger ensemble techniques. This limitation might result in less-than-ideal performance, particularly when managing big datasets or obtaining quicker training durations.

Additional presumptions include the data's stationarity, which holds that the distribution of the data stays constant across time. In practical situations, variables like modifications to traffic laws or road conditions may cause data distributions to change, which might eventually impair model performance.

Results

4.1 Overview

In this chapter, the outcomes of training and testing machine learning models for classifying the severity of traffic accidents are presented. To determine the effectiveness of the models, performance is evaluated using measures like precision, recall, F1-score, and overall accuracy.

The results are arranged in the following manner: Section 4.2 describes how each model performed in different experimental setups. To determine the models' advantages and disadvantages, a comparative analysis is presented in Section 4.3. Last but not least, Section 4.4 highlights the most important discoveries and discusses the results' wider ramifications.

4.2 Experimental Results

4.2.1 Main Pipeline Results

Most models perform similarly, with no single model significantly outperforming the others. The macro average (*macro avg*) metric is particularly important because it reflects the model's performance across all classes equally, regardless of class distribution. The noticeable difference between *macro avg* and *weighted avg* highlights the substantial impact of class imbalance on model performance. This suggests that, despite applying SMOTE to artificially inflate minority class data, the models still struggle to effectively learn minority class patterns.

An interesting observation is that even though Decision Tree scored the lowest in overall accu-

| Algorithm | Metric Type | F1-Score | Accuracy |
|----------------------|---------------------|---------------|---------------|
| Decision Tree | Weighted Avg | 0.7418 | 0.7272 |
| Decision Tree | Macro Avg | 0.3559 | 0.7272 |
| Random Forest | Weighted Avg | 0.7863 | 0.8223 |
| Random Forest | Macro Avg | 0.3532 | 0.8223 |
| XGBoost | Weighted Avg | 0.7781 | 0.8193 |
| XGBoost | Macro Avg | 0.3427 | 0.8193 |
| MLP | Weighted Avg | 0.7783 | 0.8206 |
| MLP | Macro Avg | 0.3345 | 0.8206 |

Table 4.1: Summary of Macro Average and Weighted Average Metrics Across Algorithms

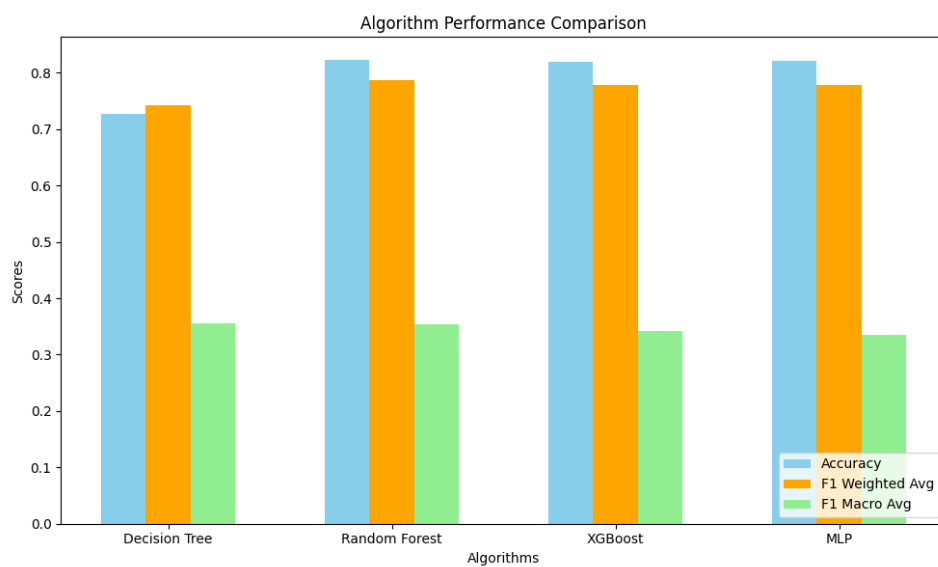


Figure 4.1: Comparison of algorithm performance across accuracy, weighted average F1-score, and macro average F1-score.

racy (0.7272), it achieved the highest *macro avg* F1-score (0.3559). This finding underscores the importance of looking beyond the accuracy metric when evaluating model performance. Decision Trees, despite their overall lower accuracy, could potentially be better at identifying severe cases, which are often of higher societal and practical importance.

4.2.2 Batch Pipeline Results

The observer assessing the batch pipeline should anticipate a thorough comparison between the baseline, or first model evaluation, and the 'improved' predictions that come after the model has been retrained on the batch data. This comparison is essential for comprehending

how the model adjusts to fresh data, indicating any improvements in its predictive power. Through the examination of the variations in performance metrics like recall, accuracy, and F1-score between the baseline and the retrained model, we may determine whether the retraining procedure has effectively addressed previous issues like overfitting or class imbalance. The examination also provides insights on the model's capacity for generalization, which improves understanding of how it reacts to novel patterns in the incoming data.

| Algorithm | Metric Type | F1-Score | Accuracy |
|---------------|--------------|----------|----------|
| Decision Tree | Weighted Avg | 0.7418 | 0.7272 |
| Decision Tree | Macro Avg | 0.3559 | 0.7272 |
| Random Forest | Weighted Avg | 0.7863 | 0.8223 |
| Random Forest | Macro Avg | 0.3532 | 0.8223 |
| XGBoost | Weighted Avg | 0.7781 | 0.8193 |
| XGBoost | Macro Avg | 0.3427 | 0.8193 |
| MLP | Weighted Avg | 0.7783 | 0.8206 |
| MLP | Macro Avg | 0.3345 | 0.8206 |

Table 4.2: *Summary of Macro Average and Weighted Average Metrics Across Algorithms, Baseline Evaluation*

| Algorithm | Metric Type | F1-Score | Accuracy |
|---------------|--------------|----------|----------|
| Decision Tree | Weighted Avg | 0.7354 | 0.7160 |
| Decision Tree | Macro Avg | 0.3443 | 0.7160 |
| Random Forest | Weighted Avg | 0.7852 | 0.8118 |
| Random Forest | Macro Avg | 0.3540 | 0.8118 |
| XGBoost | Weighted Avg | 0.7797 | 0.8085 |
| XGBoost | Macro Avg | 0.3602 | 0.8085 |
| MLP | Weighted Avg | 0.7749 | 0.7979 |
| MLP | Macro Avg | 0.3468 | 0.7979 |

Table 4.3: *Summary of Macro Average and Weighted Average Metrics Across Algorithms, Post-Retraining Evaluation*

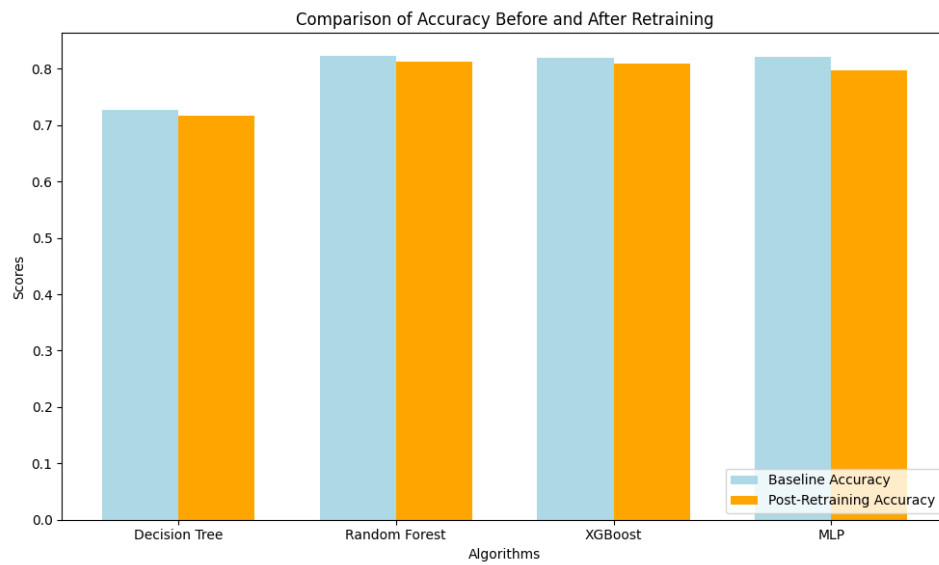


Figure 4.2: Accuracy comparison.

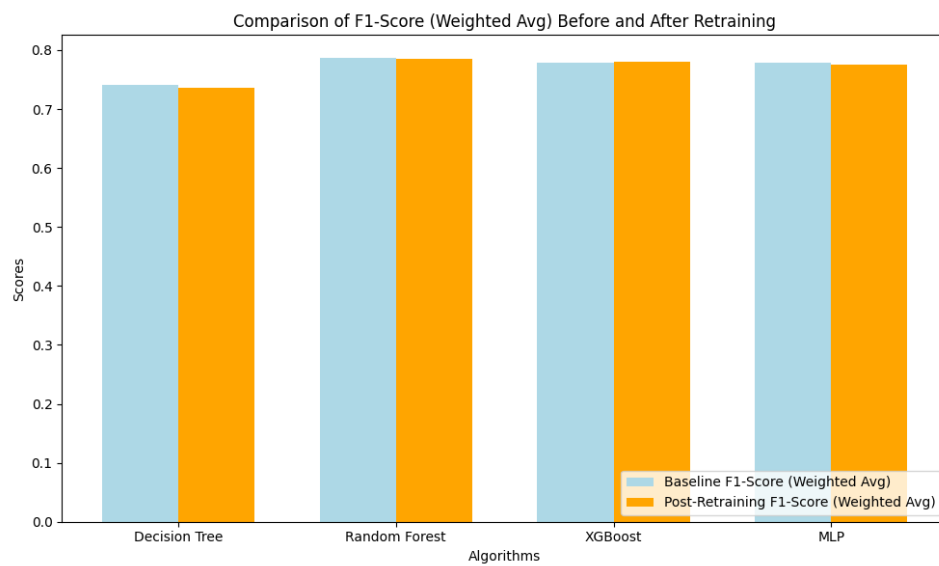


Figure 4.3: Weighted F1-score comparison.

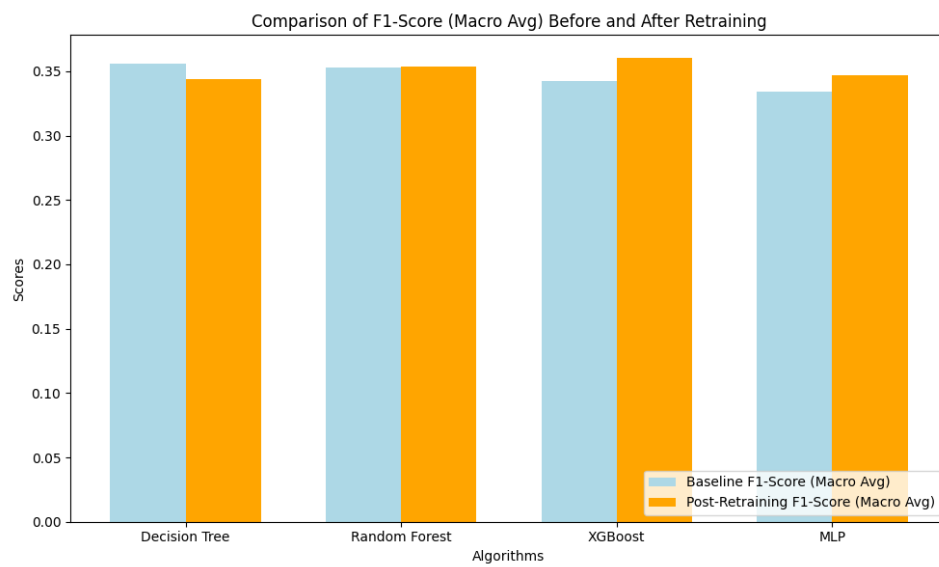


Figure 4.4: *Macro F1-score comparison.*

4.3 Comparative Analysis

Different performance patterns for the four algorithms—Decision Tree, Random Forest, XGBoost, and Multi-Layer Perceptron (MLP)—arose in the first evaluation of the main pipeline, in which the models were trained only on the original dataset without any additional retraining on batch data.

With a weighted F1-score of 0.7863 and an accuracy of 0.8223, Random Forest performed best across all criteria. With similar weighted F1-scores of 0.7781 and 0.7783, respectively, and accuracy ratings just over 0.8190, XGBoost and MLP came in second and third. The Decision Tree method, on the other hand, performed poorly, achieving an accuracy of 0.7272 and a weighted F1-score of 0.7418.

The macro average scores reveal a different perspective. Despite Random Forest's somewhat higher macro F1-score of 0.3532 than that of XGBoost and MLP, all models' total macro averages were far lower than those of their weighted counterparts. This indicates a struggle to maintain consistent performance across all classes, particularly for the minority classes with fewer samples.

Models primarily perform better on the majority class (Slight severity) at the expense of minority classes (Serious and Fatal), as evidenced by the discrepancy between weighted and macro F1-scores. Decision Tree showed limited effectiveness in managing class imbalance, but it outperformed Random Forest by a small margin with a macro F1-score of 0.3559.

Following retraining on batch data, performance metrics exhibited varied adjustments. Random Forest maintained its superiority over other models, with a slightly lower weighted F1-score of 0.7852 and accuracy of 0.8118, highlighting the impact of the class imbalance and the challenge of ensuring class distribution consistency in new data batches. With an accuracy of 0.8085 and a weighted F1-score of 0.7797, XGBoost likewise saw a slight decrease. MLP had a more noticeable decline in performance indicators after retraining, with an accuracy of 0.7979 and a weighted F1-score of 0.7749.

Nonetheless, XGBoost showed a 5.1% improvement in its macro F1-score, rising from 0.3427

to 0.3602. MLP also displayed a smaller 3.7% increase in macro F1-score, indicating improved minority class management. The weighted and macro F1-scores of Decision Tree, on the other hand, dropped to 0.7354 and 0.3443, respectively, demonstrating persistent underperformance. Meanwhile, Random Forest remained relatively stable, with a slight decrease in accuracy from 0.8223 to 0.8118 and a minor drop in weighted F1-score from 0.7863 to 0.7852, while its macro F1-score increased marginally from 0.3532 to 0.3540. This suggests consistent overall performance despite minor fluctuations.

4.4 Summary of Key Findings

This chapter presents a summary of the key findings from the comparative analysis of the model performance before and after retraining on batch data. The primary goal was to evaluate how retraining influenced the accuracy, F1-scores, and the models' ability to handle imbalanced class distributions.

Upon retraining, different models exhibited various responses to the batch data. XGBoost displayed the most significant improvement, as evidenced by a 5.1% rise in its macro F1-score, indicating an improvement in the model's capacity to manage the minority class. MLP demonstrated a lesser enhancement of 3.7%, reflecting a beneficial development in tackling the imbalance, but this was not as pronounced as that of XGBoost. In contrast, the Decision Tree model did not perform as well after retraining; its weighted and macro F1-scores dropped, indicating that the additional data did not benefit it as much. Random Forest's performance showed only minor variations in accuracy and F1-scores, demonstrating stability even after retraining.

- **XGBoost:** The best-performing model, showing a 5.1% improvement in macro F1-score, proving its robustness in improving predictions for the minority class.
- **MLP:** A more modest improvement of 3.7% in macro F1-score suggests some positive adjustment, particularly for the underrepresented classes.
- **Random Forest:** Maintained stable performance, with minimal changes across metrics,

showing that retraining on batch data did not significantly alter its ability to generalize.

- **Decision Tree:** Experienced a decline in both accuracy and F1-scores, which suggests it may be too simplistic for handling complex class imbalances and is less effective with additional data.

To summarize, retraining the models on batch data revealed that XGBoost and MLP were the most adaptable, with XGBoost exhibiting the greatest performance improvements. Random Forest maintained its stability, whereas the Decision Tree model did not show improvement and actually exhibited a decline in performance. The results imply that more sophisticated models, such as XGBoost, are more appropriate for dealing with imbalanced datasets and retraining processes, while simpler models may need additional tuning or adjustments.

Conclusions

5.1 Summary of Research and Findings

The purpose of this study was to examine how effective various machine-learning algorithms are at predicting the severity of traffic accidents, concentrating specifically on assessing the effect of model retraining with extra batch data on performance. The models taken into account were Decision Tree, Random Forest, XGBoost, and MLP. During the study, various important metrics, such as accuracy and F1-scores (both weighted and macro averages), were examined before and after retraining.

The results showed significant differences in how the models responded to retraining. The most notable enhancement was observed in XGBoost, which experienced a macro F1-score rise of 5.1%. This suggests that post-retraining, the model became more adept at managing imbalanced data. The macro F1-score of MLP also improved modestly, with an increase of 3.7%. Conversely, the performance of Decision Tree after retraining was lacking, as evidenced by declines in both accuracy and F1-scores. The performance of Random Forest remained mostly stable, with minor variations in accuracy and F1-scores, indicating that retraining did not have a significant impact. The results demonstrate that more intricate models, such as XGBoost, tend to benefit more from retraining than simpler models, which may need additional fine-tuning or modifications.

5.2 Contributions and Significance

This study highlights the essential contribution of MLOps practices to the research process, alongside its main findings. Utilizing MLOps, especially with resources such as automated CI/CD pipelines, model versioning, and continuous monitoring, facilitated a more efficient process for model retraining and evaluation. The ability to quickly iterate on model performance using automated workflows enabled the rapid iteration on model performance and efficient incorporation of new data. This helped ensure consistent and scalable enhancements to model accuracy and F1-scores.

By incorporating MLOps principles, this research was able to effectively tackle the challenges of deploying and maintaining machine learning models, especially when retraining on batch data. Utilizing MLOps facilitated the seamless incorporation of the retraining pipeline into the current workflow, enhancing process reliability and efficiency. This shows the practical benefits of MLOps in guaranteeing that machine learning systems are effective and sustainable in real-world applications, particularly when working with large, imbalanced datasets.

5.3 Practical Implications and Recommendations

MLOps can revolutionize the deployment of machine learning models at scale, particularly when ongoing retraining is required. Organizations operating in fields such as traffic forecasting, where real-time data alterations and model precision are vital, can benefit from MLOps principles to guarantee ongoing model updates without the need for manual involvement. As demonstrated in this research, automated performance tracking and retraining would enable organizations to rapidly adjust to new traffic patterns, thereby enhancing model relevance and safety outcomes.

Thus, adopting MLOps practices—such as pipeline automation, model versioning, and regular performance monitoring—should be a core recommendation for teams seeking to deploy models that need continuous learning, ensuring models can be maintained at optimal performance levels in dynamic environments.

5.4 Limitations and Suggestions for Future Research

This study offers valuable insights into the performance of different machine learning algorithms and the application of MLOps in model training and retraining. However, it has several limitations that future research can address.

One key limitation is the dataset used in this research. The dataset, while comprehensive, is imbalanced. This imbalance impacted the performance of the selected algorithms, especially regarding minority classes. Even though SMOTE was leveraged in this study, future research may investigate novel approaches to address this imbalance. Furthermore, choosing a dataset with a better balance would enhance the model's generalization capabilities.

Furthermore, the algorithms evaluated in this study could be broadened to encompass deeper or more sophisticated models. Investigating more sophisticated algorithms, such as deep learning models or hybrid methods, may yield better performance, especially in the context of large-scale or real-time data. Supporting GPU acceleration could greatly reduce training times, making it feasible to use deeper, more computationally intensive algorithms without affecting performance.

Another aspect that could be enhanced is the infrastructure that underpins model training and evaluation. Implementing a runner autoscaling group to handle concurrent algorithm training and evaluation could address resource limitations and allow the system to scale efficiently based on demand. This would be especially advantageous in production environments where several algorithms must be trained or assessed at the same time.

Future research in MLOps and continuous integration should explore restructuring pipelines to center around GitHub Actions (GHA) for improved integration and automated model deployment. Utilizing GHA allows teams to integrate versioning, automated testing, and deployment into their machine learning pipelines more effortlessly, leading to a smoother workflow and an accelerated feedback loop.

Additionally, incorporating code quality scans into the pipeline would improve the overall reliability and maintainability of the code, thereby decreasing the likelihood of errors throughout

the model development lifecycle. This would represent a significant advance in refining the processes for deploying models in production environments.

Automating the detection of data drift is another aspect that could enhance future efforts as part of ongoing improvement. Data drift, which occurs when the statistical properties of data evolve over time, can have a significant effect on the performance of machine learning models. To ensure that models remain accurate in environments characterized by change and dynamism, it would be beneficial to implement automated tools for real-time detection and remediation of data drift.

As a final point, it is recommended that future studies create a strong CI/CD pipeline to facilitate the continuous deployment of updated models via a web server. This would enable real-time updates and retraining of models, guaranteeing that deployed models align with the latest data, which improves performance and responsiveness in production.

5.5 Final Statement

Valuable insights into the practical application of machine learning algorithms, especially regarding traffic accident severity classification, have been provided by this research. This study has demonstrated how model performance can be systematically enhanced through retraining processes, monitoring, and automation by comparing different algorithms and applying MLOps principles. Utilizing MLOps allowed us to create machine learning pipelines that are more robust, scalable, and reproducible, thereby facilitating ongoing enhancements in model accuracy and performance.

This work has wider implications beyond just traffic prediction models; it underscores the potential of MLOps to transform how machine learning models are deployed, maintained, and updated. Organizations can guarantee their models stay attuned to new data—leading to improved decision-making and resource distribution in vital areas—by automating deployment, performance evaluation, and retraining. In a sector as important as road safety, high-quality up-to-date models and predictions have the potential to prevent indescribable human suffering.

To sum up, this research highlights the significance of integrating machine learning with operational practices. This research contributes to the creation of predictive models with societal impact and lays a strong groundwork for enhancing MLOps practices in future endeavors. The results achieved here lay the groundwork for a more effective, efficient, and adaptive machine learning lifecycle, paving the way for future innovations in automated model retraining and deployment.

Bibliography

- Agrawal, Himanshu (2023). *Kubernetes Fundamentals: A Step-by-Step Development and Interview Guide*. English. 1;1st; Berkeley, CA: Apress. ISBN: 1484297296;9781484297292;9781484297285;1
- Ahmed, Mohammed I. (2024). *Cloud-Native DevOps: Building Scalable and Reliable Applications*. English. 1st ed. Berkeley, CA: Apress L. P. ISBN: 9798868804069;9798868804076;
- Alhaek, Fares, Liang, Weichao, Rajeh, Taha M., Javed, Muhammad Hafeez, and Li, Tianrui (Feb. 2024). "Learning spatial patterns and temporal dependencies for traffic accident severity prediction: A deep learning approach". In: *Knowledge-Based Systems* 286. ISSN: 09507051. DOI: 10.1016/j.knosys.2024.111406.
- Ankita, Mittal, Sonam, Sharma, Ishu, and Kumar, Atul (2023). "Comparative Analysis of Shallow Learning and Deep Learning". In: *2023 International Conference on Next Generation Electronics, NEleX 2023*. Institute of Electrical and Electronics Engineers Inc. DOI: 10.1109/NEleX59773.2023.10421385.
- Baldini, Gianmarco and Cerutti, Isabella (2023). "Classification of Optical Transmission Anomalies with Convolutional Neural Networks and 2D Histograms". In: *2023 IEEE International Mediterranean Conference on Communications and Networking, MeditCom 2023*. Institute of Electrical and Electronics Engineers Inc., pp. 62–67. ISBN: 9798350333732. DOI: 10.1109/MeditCom58224.2023.10266654.
- Bao, Chun, Chen, Jianqiu, Xiong, Huan, Cao, Xiali, Wang, Shiyu, Liu, Bote, Lou, Benxiao, and Gu, Guobin (2024). "Research on Classification and Prediction of General Traffic Ac-

- cidents on National and Provincial Highways Based on BP Neural Networks”. In: *2024 4th International Conference on Neural Networks, Information and Communication Engineering, NNICE 2024*. Institute of Electrical and Electronics Engineers Inc., pp. 781–784. DOI: 10.1109/NNICE61279.2024.10498278.
- Boorshtein, Marc, Surovich, Scott, and Price, Ed (2024). *Kubernetes - an Enterprise Guide: Master Containerized Application Deployments, Integrate Enterprise Systems, and Achieve Scalability*. English. 3rd ed. Birmingham: Packt Publishing, Limited. ISBN: 9781835086957;1835086950;
- Bougna, Théophile, Hundal, Gursmeep, and Taniform, Peter (2022). “Quantitative Analysis of the Social Costs of Road Traffic Crashes Literature”. In: *Accident Analysis & Prevention* 165, p. 106282. ISSN: 0001-4575. DOI: <https://doi.org/10.1016/j.aap.2021.106282>. URL: <https://www.sciencedirect.com/science/article/pii/S0001457521003134>.
- Burch, Cynthia, Cook, Lawrence, and Dischinger, Patricia (2014). “A Comparison of KABCO and AIS Injury Severity Metrics Using CODES Linked Data”. In: *Traffic Injury Prevention* 15 (6), pp. 627–630. ISSN: 1538957X. DOI: 10.1080/15389588.2013.854348.
- Cuquantum, T. (2024). *Data Analysis Foundations with Python*. English. Packt Publishing.
- Department for Transport (2024). *Guide to severity adjustments for reported road casualties Great Britain*. Updated: 28 Nov 2024, Accessed: 24 Dec 2024. URL: <https://www.gov.uk/government/publications/guide-to-severity-adjustments-for-reported-road-casualty-statistics/guide-to-severity-adjustments-for-reported-road-casualties-great-britain>.
- Esppenchutz, Gláucia (2023). “Automating Your Data Ingestion Pipelines”. English. In: *Data Ingestion with Python Cookbook*. United Kingdom: Packt Publishing, Limited.
- Galli, Soledad (2024). *Python Feature Engineering Cookbook - Third Edition*. English. Packt Publishing.

- Garg, Satvik, Pundir, Pradyumn, Rathee, Geetanjali, Gupta, P. K., Garg, Somya, and Ahlawat, Saransh (2021). "On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps". In: *Proceedings - 2021 IEEE 4th International Conference on Artificial Intelligence and Knowledge Engineering, AIKE 2021*. Institute of Electrical and Electronics Engineers Inc., pp. 25–28. DOI: 10.1109/AIKE52691.2021.00010.
- Glassner, Andrew (2021). *Deep learning: a visual approach*. English. 1st ed. San Francisco, Calif: No Starch Press.
- Gupta, Saumya, Bhatia, Madhulika, Memoria, Meenakshi, and Manani, Preeti (2022). "Prevalence of GitOps, DevOps in Fast CI/CD Cycles". In: *2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON)*. Vol. 1, pp. 589–596. DOI: 10.1109/COM-IT-CON54601.2022.9850786.
- Gupta, Utkarsh, Varun, M. K., and Srinivasa, Gowri (2022). "A Comprehensive Study of Road Traffic Accidents: Hotspot Analysis and Severity Prediction Using Machine Learning". In: *IBSSC 2022 - IEEE Bombay Section Signature Conference*. ML in traffic accident prediction Relevant features. Institute of Electrical and Electronics Engineers Inc. DOI: 10.1109/IBSSC56953.2022.10037449.
- Hattori, Yuki (2024). *DevOps Unleashed with Git and GitHub: Automate, Collaborate, and Innovate to Enhance Your DevOps Workflow and Development Experience*. English. 1st ed. Birmingham: Packt Publishing, Limited.
- Haviv, Yaron and Gift, Noah (2023). *Implementing MLOps in the Enterprise*. English. 1st. O'Reilly Media, Inc.
- Hazaymeh, Khaled, Almagbile, Ali, and Alomari, Ahmad H. (Apr. 2022). "Spatiotemporal Analysis of Traffic Accidents Hotspots Based on Geospatial Techniques". In: *ISPRS International Journal of Geo-Information* 11 (4). ISSN: 22209964. DOI: 10.3390/ijgi11040260.

- Islam, Md Kamrul, Reza, Imran, Gazder, Uneb, Akter, Rocksana, Arifuzzaman, Md, and Rahman, Muhammad Muhitur (Nov. 2022). "Predicting Road Crash Severity Using Classifier Models and Crash Hotspots". In: *Applied Sciences (Switzerland)* 12 (22). ISSN: 20763417. DOI: 10.3390/app122211354.
- Kamh, Hussien, Alyami, Saleh H., Khattak, Afaq, Alyami, Mana, and Almujiabah, Hamad (2024). "Exploring Road Traffic Accidents Hotspots Using Clustering Algorithms and GIS-based Spatial Analysis". English. In: *IEEE access*, pp. 1–1.
- Karimi, Esmail, Haghighi, Farshidreza, Sheykhfard, Abbas, Azmoodeh, Mohammad, and Shaaban, Khaled (Mar. 2023). "Self-Organized Neural Network Method to Identify Crash Hotspots". In: *Future Transportation* 3 (1), pp. 286–295. ISSN: 26737590. DOI: 10.3390/futuretransp3010017.
- Karunarathne, M. A.W., Wijayanayake, W. M.J.I., and Prasadika, A. P.K.J. (2024). "DevOps Adoption in Software Development Organizations: A Systematic Literature Review". In: *ICARC 2024 - 4th International Conference on Advanced Research in Computing: Smart and Innovative Trends in Next Generation Computing Technologies*. Institute of Electrical and Electronics Engineers Inc., pp. 282–287. DOI: 10.1109/ICARC61713.2024.10499789.
- Kaufmann, Michael (2024). *GitHub Actions Cookbook: A Practical Guide to Automating Repetitive Tasks and Streamlining Your Development Process*. English. 1st ed. Birmingham: Packt Publishing, Limited. ISBN: 9781835468944;1835468942;
- Kreuzberger, Dominik, Kühl, Niklas, and Hirschl, Sebastian (2023). "Machine Learning Operations (MLOps): Overview, Definition, and Architecture". In: *IEEE Access* 11, pp. 31866–31879. DOI: 10.1109/ACCESS.2023.3262138.
- Laster, Brent (2023). *Learning GitHub Actions: Automation and Integration of CI/CD With GitHub*. English. 1st. O'Reilly. ISBN: 9781098131074;109813107X;

- Le, Khanh Giang, Liu, Pei, and Lin, Liang Tay (2022). "Traffic accident hotspot identification by integrating kernel density estimation and spatial autocorrelation analysis: a case study". In: *International Journal of Crashworthiness* 27 (2), pp. 543–553. ISSN: 17542111. DOI: 10.1080/13588265.2020.1826800.
- Liu, Yong (2022). "Tracking Code and Data Versioning". English. In: *Practical Deep Learning at Scale with MLflow*. United Kingdom: Packt Publishing, Limited.
- Mäkinen, Sasu (2021). *Designing an open-source cloud-native MLOps pipeline*. URL: <http://www.cs.helsinki.fi/>.
- Matloff, Norman (2024). *The Art of Machine Learning: A Hands-On Guide to Machine Learning with R*. English. 1st ed. New York: No Starch Press.
- Migliaccio, Alessandro and Iannone, Giovanni (2023). *Systems Engineering Neural Networks*. 1st ed. Wiley. ISBN: 9781119901990;1119901995;9781119902003;1119902002;
- Ngo-Ho, Anh Khoi, Bui, Hoang Bac, and Pham, Van Trieu (2023). "Continuous Learning for Automatic Identification of OC EO Antique Glass Jewelry". In: *Proceedings - International Conference on Knowledge and Systems Engineering, KSE*. Institute of Electrical and Electronics Engineers Inc. DOI: 10.1109/KSE59128.2023.10299466.
- Nikhil Ketkar, Jojo M. (2021). *Deep Learning with Python - Learn Best Practices of Deep Learning Models with PyTorch (2nd Edition)*. English. 2;2nd;Second; Berkeley, CA: Apress, an imprint of Springer Nature.
- Pacheco, Lorena Barreto Simedo, Rahman, Musfiqur, Rabbi, Fazle, Fathollahzadeh, Pouya, Abdellatif, Ahmad, Shihab, Emad, Chen, Tse Hsun, Yang, Jinqiu, and Zou, Ying (Apr. 2024). "DVC in open source ml-development: The action and the reaction". In: *Proceedings - 2024 IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, CAIN 2024*. Association for Computing Machinery, Inc, pp. 75–80. DOI: 10.1145/3644815.3644965.

- Parashar, Anubha, Parashar, Apoorva, Ding, Weiping, Shabaz, Mohammad, and Rida, Imad (Aug. 2023). "Data preprocessing and feature selection techniques in gait recognition: A comparative study of machine learning and deep learning approaches". In: *Pattern Recognition Letters* 172, pp. 65–73. ISSN: 01678655. DOI: 10.1016/j.patrec.2023.05.021.
- Popuri, Anoushka and Miller, John (2023). "Generative Adversarial Networks in Image Generation and Recognition". In: *Proceedings - 2023 International Conference on Computational Science and Computational Intelligence, CSCI 2023*. Institute of Electrical and Electronics Engineers Inc., pp. 1294–1297. ISBN: 9798350361513. DOI: 10.1109/CSCI62032.2023.00212.
- Reddy, J. Saikiran, Padal, S. Durga Prashanth, Mayan, J. Albert, Catharine, A., and Shan-thamalar, J. Jeslin (2024). "Efficient Application Deployment: GitOps for Faster and Secure CI/CD Cycles". In: *2024 International Conference on Advances in Modern Age Technologies for Health and Engineering Science, AMATHE 2024*. Institute of Electrical and Electronics Engineers Inc. DOI: 10.1109/AMATHE61652.2024.10582118.
- Roh, Seol, Jeong, Ki Moon, Cho, Hye Young, and Huh, Eui Nam (2023). "An Efficient Microservices Architecture for MLOps". In: *International Conference on Ubiquitous and Future Networks, ICUFN*. Vol. 2023-July. IEEE Computer Society, pp. 652–654. DOI: 10.1109/ICUFN57995.2023.10201181.
- Santos, Daniel, Saias, José, Quaresma, Paulo, and Nogueira, Vítor Beires (Dec. 2021). "Machine learning approaches to traffic accident analysis and hotspot prediction". In: *Computers* 10 (12). ISSN: 2073431X. DOI: 10.3390/computers10120157.
- Sayfan, Gigi (2023). *Mastering Kubernetes: Dive into Kubernetes and learn how to create and operate world-class cloud-native systems*. English. 4th ed. Birmingham: Packt Publishing. ISBN: 1804611395;9781804611395;
- Shiri, Farhad Morteza pour, Perumal, Thinagaran, Mustapha, Norwati, and Mohamed, Raihani (2024). *A Comprehensive Overview and Comparative Analysis on Deep Learning Models:*

CNN, RNN, LSTM, GRU. arXiv: 2305.17473 [cs.LG]. URL: <https://arxiv.org/abs/2305.17473>.

Singh, Pradeep (2022). *Fundamentals and Methods of Machine and Deep Learning*. Wiley-Scrivener. ISBN: 9781119821250;1119821258;

Teixeira, Rafael, Almeida, Leonardo, Rodrigues, Pedro, Antunes, Mario, Gomes, Diogo, and Aguiar, Rui L. (2024). "Shallow vs. Deep Learning: Prioritizing Efficiency in Next Generation Networks". In: *Proceedings - 2024 11th International Conference on Future Internet of Things and Cloud, FiCloud 2024*. Institute of Electrical and Electronics Engineers Inc., pp. 308–315. DOI: 10.1109/FiCloud62933.2024.00055.

Testi, Matteo, Ballabio, Matteo, Frontoni, Emanuele, Iannello, Giulio, Moccia, Sara, Soda, Paolo, and Vessio, Gennaro (2022). *MLOps: A Taxonomy and a Methodology*. DOI: 10.1109/ACCESS.2022.3181730.

Zarate, Gorka, Minon, Raul, Diaz-De-Arcaya, Josu, and Torre-Bastida, Ana I. (2022). "K2E: Building MLOps Environments for Governing Data and Models Catalogues while Tracking Versions". In: *2022 IEEE 19th International Conference on Software Architecture Companion, ICSA-C 2022*. Institute of Electrical and Electronics Engineers Inc., pp. 206–209. DOI: 10.1109/ICSA-C54293.2022.00047.

Zhang, Xu, Liu, Lin, Xiao, Luzi, and Ji, Jiakai (2020). "Comparison of machine learning algorithms for predicting crime hotspots". In: *IEEE Access* 8, pp. 181302–181310. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3028420.

Appendix

A.1 Classification Reports

A.1.1 Main Pipeline

Table A.1: *Classification Report for Decision Tree Classifier*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|-----------|--------|----------|---------|
| 0 (Fatal) | 0.0360 | 0.0569 | 0.0441 | 1423 |
| 1 (Serious) | 0.1642 | 0.2073 | 0.1833 | 14122 |
| 2 (Slight) | 0.8625 | 0.8192 | 0.8403 | 90164 |
| Macro Avg | 0.3542 | 0.3611 | 0.3559 | 105709 |
| Weighted Avg | 0.7581 | 0.7272 | 0.7418 | 105709 |
| Accuracy | 0.7272 | | | |

Table A.2: *Classification Report for Random Forest Classifier*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|-----------|--------|----------|---------|
| 0 (Fatal) | 0.0730 | 0.0253 | 0.0376 | 1423 |
| 1 (Serious) | 0.2155 | 0.0828 | 0.1197 | 14122 |
| 2 (Slight) | 0.8590 | 0.9506 | 0.9025 | 90164 |
| Macro Avg | 0.3825 | 0.3529 | 0.3532 | 105709 |
| Weighted Avg | 0.7624 | 0.8223 | 0.7863 | 105709 |
| Accuracy | 0.8223 | | | |

Table A.3: *Classification Report for XGBoost Classifier*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0565 | 0.1286 | 0.0785 | 1423 |
| 1 (Serious) | 0.1680 | 0.0265 | 0.0458 | 14122 |
| 2 (Slight) | 0.8584 | 0.9544 | 0.9039 | 90164 |
| Macro Avg | 0.3610 | 0.3698 | 0.3427 | 105709 |
| Weighted Avg | 0.7554 | 0.8193 | 0.7781 | 105709 |
| Accuracy | 0.8193 | | | |

Table A.4: *Classification Report for MLP*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0384 | 0.0654 | 0.0483 | 1423 |
| 1 (Serious) | 0.1642 | 0.0306 | 0.0516 | 14122 |
| 2 (Slight) | 0.8566 | 0.9562 | 0.9036 | 90164 |
| Macro Avg | 0.3530 | 0.3507 | 0.3345 | 105709 |
| Weighted Avg | 0.7530 | 0.8206 | 0.7783 | 105709 |
| Accuracy | 0.8206 | | | |

A.1.2 Batch Pipeline

Decision Tree Classifier

Table A.5: *Decision Tree - Evaluation Baseline*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|-----------|--------|----------|---------|
| 0 (Fatal) | 0.0360 | 0.0569 | 0.0441 | 1423 |
| 1 (Serious) | 0.1642 | 0.2073 | 0.1833 | 14122 |
| 2 (Slight) | 0.8625 | 0.8192 | 0.8403 | 90164 |
| Macro Avg | 0.3542 | 0.3611 | 0.3559 | 105709 |
| Weighted Avg | 0.7581 | 0.7272 | 0.7418 | 105709 |
| Accuracy | 0.7272 | | | |

Table A.6: *Decision Tree - Evaluation after retraining*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|-----------|--------|----------|---------|
| 0 (Fatal) | 0.0164 | 0.0297 | 0.0212 | 269 |
| 1 (Serious) | 0.1561 | 0.2115 | 0.1796 | 3016 |
| 2 (Slight) | 0.8619 | 0.8044 | 0.8321 | 19301 |
| Macro Avg | 0.3448 | 0.3485 | 0.3443 | 22586 |
| Weighted Avg | 0.7576 | 0.7160 | 0.7354 | 22586 |
| Accuracy | 0.7160 | | | |

Random Forest Classifier**Table A.7:** *Random Forest - Evaluation Baseline*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0730 | 0.0253 | 0.0376 | 1423 |
| 1 (Serious) | 0.2155 | 0.0828 | 0.1197 | 14122 |
| 2 (Slight) | 0.8590 | 0.9506 | 0.9025 | 90164 |
| Macro Avg | 0.3825 | 0.3529 | 0.3532 | 105709 |
| Weighted Avg | 0.7624 | 0.8223 | 0.7863 | 105709 |
| Accuracy | 0.8223 | | | |

Table A.8: *Random Forest - Evaluation after retraining*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0312 | 0.0149 | 0.0202 | 269 |
| 1 (Serious) | 0.2127 | 0.1114 | 0.1462 | 3016 |
| 2 (Slight) | 0.8619 | 0.9323 | 0.8957 | 19301 |
| Macro Avg | 0.3686 | 0.3529 | 0.3540 | 22586 |
| Weighted Avg | 0.7653 | 0.8118 | 0.7852 | 22586 |
| Accuracy | 0.8118 | | | |

XGBoost Classifier**Table A.9:** *XGBoost - Evaluation Baseline*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0565 | 0.1286 | 0.0785 | 1423 |
| 1 (Serious) | 0.1680 | 0.0265 | 0.0458 | 14122 |
| 2 (Slight) | 0.8584 | 0.9544 | 0.9039 | 90164 |
| Macro Avg | 0.3610 | 0.3698 | 0.3427 | 105709 |
| Weighted Avg | 0.7554 | 0.8193 | 0.7781 | 105709 |
| Accuracy | 0.8193 | | | |

Table A.10: *XGBoost - Evaluation after retraining*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0631 | 0.1822 | 0.0937 | 269 |
| 1 (Serious) | 0.1956 | 0.0584 | 0.0899 | 3016 |
| 2 (Slight) | 0.8626 | 0.9345 | 0.8971 | 19301 |
| Macro Avg | 0.3737 | 0.3917 | 0.3602 | 22586 |
| Weighted Avg | 0.7640 | 0.8085 | 0.7797 | 22586 |
| Accuracy | 0.8085 | | | |

MLP Tables**Table A.11:** *MLP - Evaluation Baseline*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0384 | 0.0654 | 0.0483 | 1423 |
| 1 (Serious) | 0.1642 | 0.0306 | 0.0516 | 14122 |
| 2 (Slight) | 0.8566 | 0.9562 | 0.9036 | 90164 |
| Macro Avg | 0.3530 | 0.3507 | 0.3345 | 105709 |
| Weighted Avg | 0.7530 | 0.8206 | 0.7783 | 105709 |
| Accuracy | 0.8206 | | | |

Table A.12: *MLP - Evaluation after retraining*

| Class | Precision | Recall | F1-Score | Support |
|---------------------|------------------|---------------|-----------------|----------------|
| 0 (Fatal) | 0.0298 | 0.0520 | 0.0379 | 269 |
| 1 (Serious) | 0.1753 | 0.0845 | 0.1141 | 3016 |
| 2 (Slight) | 0.8592 | 0.9197 | 0.8884 | 19301 |
| Macro Avg | 0.3547 | 0.3521 | 0.3468 | 22586 |
| Weighted Avg | 0.7580 | 0.7979 | 0.7749 | 22586 |
| Accuracy | 0.7979 | | | |