[BKR+22] N. Baumann, E. Kusmenko, J. Ritz, B. Rumpe, M. B. Weber:
Dynamic Data Management for Continuous Retraining.
In: Proceedings of MODELS 2022. Workshop MDE Intelligence, ACM, Oktber 2022.
www.se-rwth.de/publications/

Dynamic Data Management for Continuous Retraining

Nils Baumann, Evgeny Kusmenko, Jonas Ritz, Bernhard Rumpe, Moritz Benedikt Weber
{kusmenko,ritz,rumpe}@se-rwth.de
RWTH Aachen University
Aachen, Germany

ABSTRACT

Managing dynamic datasets intended to serve as training data for a Machine Learning (ML) model often emerges as very challenging, especially when data is often altered iteratively and already existing ML models should pertain to the data. For example, this applies when new data versions arise from either a generated or aggregated extension of an existing dataset a model has already been trained on. In this work, it is investigated on how a model-based approach for these training data concerns can be provided as well as how the complete process, including the resulting training and retraining process of the ML model, can therein be integrated. Hence, model-based concepts and the implementation are devised to cope with the complexity of iterative data management as an enabler for the integration of continuous retraining routines. With Deep Learning techniques becoming technically feasible and massively being developed further over the last decade, MLOps, aiming to establish DevOps tailored to ML projects, gained crucial relevance. Unfortunately, data-management concepts for iteratively growing datasets with retraining capabilities embedded in a model-driven ML development methodology are unexplored to the best of our knowledge. To fill in this gap, this contribution provides such agile data management concepts and integrates them and continuous retraining into the model-driven ML Framework MontiAnna [18]. The new functionality is evaluated in the context of a research project where ML is exploited for the optimal design of lattice structures for crash applications.

CCS CONCEPTS

• **Software and its engineering** → *Application specific development environments*; • **Computing methodologies** → *Machine learning*.

KEYWORDS

model-driven engineering, artificial intelligence, data management, retraining

This research has partly received funding from the Federal Ministry for Economic Affairs and Climate Action (BMWK) in a project called KI-LaSt under grant no. 19I21036F. The responsibility for the content of this publication is with the authors.

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3561568

ACM Reference Format:
Nils Baumann, Evgeny Kusmenko, Jonas Ritz, Bernhard Rumpe, Moritz Benedikt Weber. 2022. Dynamic Data Management for Continuous Retraining. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3550356.3561568

1 INTRODUCTION

In the past years, ML applications have been established in multiple domains reaching from healthcare applications [23] over finance [9] to sentiment analysis [6]. However, the key element to success in each ML task remains the dataset, as its quality massively impacts the prediction capability of the model [14]. Thus, intelligent management and the attached training routines of different versions can accelerate, simplify and enhance the development process and model quality tremendously. Easy ways for juggling different data versions should exist, while versioning concepts should be clear and concise. Conflicts between different dataset versions are to be avoided and the correlation from a trained model to the dataset version should be drawn.

Often the training database continuously grows even after the first version of an ML model was designed, trained, and even deployed. This happens, for example, if more data is collected in a social network by storing user interactions or if an extension is requested to enable a more precise prognosis or a prediction inside an area the original dataset does not achieve cover. Thus, it is required that starting from a base dataset to train the first ML model, it is allowed to create an extension that can then be connected to the base dataset. This means the dataset model must provide capabilities to include a reference to a base dataset or to another extension inside the extension itself. In this way, the data scientist can iteratively provide data extensions with a reference to the base datasets or other extensions and the affiliation becomes clear.

Consequently, the *ML Engineer* is left with the opportunity to improve the network's predictions with these extensions, if a model trained on the base dataset already exists. Accordingly, a retraining routine goes hand in hand with these iterative enlargements, so that the models do not always have to be trained with the whole dataset again. The importance of retraining routines especially becomes apparent when taking into account cases, where retraining with the most recent data is indispensable. Consider the case of a concept drift [22], where the distribution of the data changes over time and can cause already trained models to be less accurate. Concept drifts can have multiple reasons, for example, technology innovation, as it can be seen in our mobile phone usage changing from mainly audio calls to mainly using mobile internet. Retraining counteracts concept drifts, as the ML model is always opened up for improvement and changes through the training on extensions. This

particularly takes effect if the complete process is automized; all required information lies within the dataset, which means, that the deployment of an extension can cause a tool-chain to be triggered including the retraining and model deployment without any explicit interaction required by the *ML engineer*.

To realize the above-mentioned ideas, this paper presents the corresponding concepts and their implementation into the MontiAnna framework, as well as an evaluation based on a use case involving structural optimization with artificial intelligence. The remainder of the paper is structured as follows: in Section 2 we list the preliminaries for understanding the basics, while Section 3 covers related work including a variety of tools for data management and related publications. Our running example about structural optimization is depicted in Section 4. The main concepts are evolved in Section 5 to then be evaluated in Section 6. In the last Section 7, we conclude our results and give a short outlook for future work.

2 PRELIMINARIES

The concepts devised in this paper were implemented as a part of the MontiAnna framework [11, 18]. This framework enables model-driven development of ML software with a dedicated focus on neural networks. It comprises two tailormade Domain-Specific Languages (DSLs) and one generic language, which have all been developed using MontiCore [17].

(1) `CNNArchLang` is a modeling language intended for the design of neural network architectures. First, the input and the output are defined to match the dimensions given in the dataset. After that, a neural network is constructed by connecting the input to a chain of different layers leading to the output. A variety of layers, like Fully Connected Layers, Graph Convolutional Layers, Convolutional Layers, Pooling Layers, and many more are supported.
(2) `ConfLang` is a modeling language intended to specify hyperparameters related to the training routine in a JSON-like syntax. These include the number of epochs to be trained and the batch size, but also the used algorithm concerning the optimization.
(3) `TaggingLang` is a language for tagging additional information, for example, the dataset is specified in this language.

After the configuration is validated, conforming executable Python Code for different backends, such as MXNet/Gluon, Tensorflow, and Caffe2 can be generated. The generated code includes the creation of the network and the training procedure. While the framework itself is written in Java, the generated training code is completely written in Python. The execution of the trained network can be done in Python or C++. The latter is intended to facilitate seamless integration into larger Component & Connector (C&C) architectures modeled in EmbeddedMontiArc [15, 20]. With this framework, complex architectures can be constructed for the domain of the cyber-physical system, where C++ is often used due to its runtime performance and resource efficiency. Other components do not necessarily need to be ML-based. Hence, a complex structure of components being connected through ports is capable of carrying out complex tasks.

The MontiAnna framework comes with a Maven plugin, which was developed to wrap some of the functionality into clear and concise goals. For example, the code generation and calls to initiate the training procedure can be invoked by `mvn emadl:train`. What is especially crucial for this work, are the goals that exist for artifact management [4]. For each artifact, the deployment into an archive and the corresponding fetch procedure can be performed using Maven goals. Besides an archive for the source code and another one for pre-trained models, the most important archive in this context is the dataset archive. It contains training, test, and evaluation data wrapped in a jar file. The two goals `mvn emadl:deploy-dataset` and `mvn emadl:install-dataset` are accordingly the basis for this work concerning dataset management and versioning. The attached retraining routines are to be implemented into the `mvn emadl:train` goal.

3 RELATED WORK

In our related work, we distinguish between related concepts from publications in the literature and concepts derived from alternative tools, that take care of similar tasks.

3.1 Related Concepts in Literature

A case study of Software Engineering for Machine Learning is offered by Amershi et al. in [3] and was carried out in 2019. The authors observed software teams at Microsoft in their development process of ML software. In the context of our work, it is remarkable that "*discovering, managing and versioning the data needed for machine learning applications is much more complex and difficult than other types of software engineering*". Especially, the need for careful treatment of datasets with continuous changes is elucidated, while, as in our case, the changes may either arise from "*operations initiated by engineers themselves, or from [...] incoming fresh data (e.g., sensor data, user interactions)*". The versioning of such datasets serves as an enabler for reuse. Moreover, if the dataset is then additionally tagged to a model, experiment tracking can be carried out. A descriptive tag about the origination can help massively as well.

In 2017, Polyzotis et al. [26] researched "Data Management Challenges in Production Machine Learning", which the authors claim to be a complete class of problems existing in the intersection of Production ML and Data Management. The focus of their works lies in the relevant steps for processing datasets, which are split up into understanding, validating, cleaning, and enriching the datasets. The research is based on their experience in the development of a data-centric infrastructure for a ML platform at Google. Retraining is implicitly mentioned, as the serving data, defined as the data to create the predictions with, is partially fed back into the data lake of the training data and thereby used again.

One year later, Polyzotis et al. extended their publication and created a survey of Data Lifecycle Challenges in Production Machine Learning [27]. In this work, understanding remains one of the three main aspects, while the data validation and cleaning are condensed compared to the previous publication, and data enrichment is replaced by data preparation. For each aspect, literature is reviewed, open challenges are posed and lessons learned from their own experience in the development process are extracted. The main emphasis is put on large-scale ML pipelines and the actual steps for preprocessing. Leading back the serving data into the training procedure is stated as the completion of the lifecycle and

can retraining can consequently be seen as a crucial step, but the technical realization is not specified.

Agrawal et. al [1] presented a data management system specifically designed for ML datasets in 2019. According to the authors from Apple Inc., engineers involved in the ML development process are forced to care about data versioning and management on their own, as the majority of the frameworks emphasize training, experimentation, evaluation, and the deployment process, while the integration of a data system is neglected. As a solution for these problems, a data management system called MLdp is proposed, which has integrated assistance for annotation, exploration as well as data and feature engineering. It removes the silos and brings together all data storage to one single place thus simplifying consistency checks and enabling easy versioning as well as paving the way for data lineage. The data can be accessed comfortably via an interface and through the integration capabilities to some common ML frameworks. The idea of versioning and the integration of the data storage is depicted in our work as well. Continuous data injection for retraining is also mentioned in the work from Agrawal et. al, but not technically explained in detail.

The most comprehensive publication in our list is written by Boehm et al. [7] and comprises both the support by ML in database systems, but also database-inspired ML systems and ML lifecycle systems. Among other topics that are discussed, the closest to our work include data access methods and systems for data preparation. The starting point in the high-level end-to-end vision of the ML lifecycle is the source, which can come in various forms. These include data lakes, distributed file-system as Apache Hadoop (cf. Section 3.2) or DBMS. The publication demonstrates methodologies to access datasets with SQL to go over into a deeper integration of ML into database systems.

Many aspects influence our presented work and offer valuable lessons to take into account when devising our concepts. Nevertheless, there is a research gap regarding the seamless integration of data management concepts and intertwined retraining into a model-based framework for the development of ML applications. Thus, the design and creation of both the model, i.e. architecture and hyperparameters and the datasets remain holistically model-driven and the models should be traceable and linked to each other.

3.2 Related Tools

As the version control system Git is intended for versioning text-based artifacts only, mainly source code, an extension to the version control system Git called Git LFS was developed to store different versions of large data files which are affiliated to different versions of source code [8]. It works by introducing a text file in the Git repository containing a pointer to the location of the dataset in external storage. This extension is rather meant for isolated persistence of large files in general than for training data management purposes so it lacks tooling for ML Applications. Dolt [1] promises to be Git for data, in particular for MySQL databases and the tables contained in them. A selection of tools from [28] with ML focus is presented in the following.

A tool called DVC tends to eliminate these issues by keeping track of datasets that can be stored using different cloud storage providers like Google Drive, AWS S3, or Microsoft Azure Blob Storage, but also local storage or network-attached storage [5]. Different dataset versions are managed by taking advantage of the integration into Git through a text-based file containing a pointer likewise to Git LFS. Regarding the intentional development for ML application purposes, DVC for instance provides possibilities for pipeline development. It is completely programming language and framework agnostic.

MLFlow [5] [29] deals with the ML lifecycle in a larger scope shifting the focus away from data management and versioning, as it is the focus in this work, to subsequent steps in the ML Pipeline such as tracking, reproducing and deploying different experiments as well as the ML Pipeline itself. It thereby secondarily addresses the problem of data management and versioning through connecting to Databricks file system (DBFS) [13], as an enhancement of Apache Spark, being integrated into Google Cloud Platform, AWS S3, or Microsoft Azure. An alternative to Apache Spark and thereby also to Databricks is Apache Hadoop developed in 2005 [24]. It usually performs worse in terms of time consumed but better when it comes to comparing memory usage [12].

Pachyderm is an extensive framework for managing data science pipelines integrating different data version management concepts into automated ML Pipelines and experiment tracking [5]. Git inspires the data versioning techniques, but it operates other than Git by not being text-based and because of that being able to operate on large binary files in a faster way.

Databricks, Apache Spark, and Pachyderms Data versioning are dedicated to scaling for Big Data and problems with huge complexity, which is not the main focus of this work. They implement mechanisms for distributed storage and its interaction with calculations on distributed systems. Furthermore, none of those mentioned above tools consider a dedicated iterative enlargement of datasets and hence lack tooling for the opportunities arising from continuous retraining, but instead always train with a completely new dataset as they are thought of in a more general way.

4 RUNNING EXAMPLE AND REQUIREMENTS

In this section, the workflow of our running example is analyzed to derive requirements for the optimal support of managing dynamic datasets and integrating continuous retraining. Integrated into this, we find an exemplary role distribution in an ML project. An important notion is the intended abstraction of the requirements from the running example, as related use-cases should be supported and flexibility should be enabled. After the concepts to fulfill the requirements are explained in the next section, the evaluation in the penultimate section also falls back on this running example. The workflow steps are also depicted in Figure 1.

4.1 Running Example

The use-case we refer to is the core of a project about the Artifical Intelligence (AI)-based design of additively manufactured lattice structures for crash applications called KI-LaSt[2]. KI-LaSt aims at reducing the CO_2 emission by minimizing the weight of energy absorbing components in cars in case of an accident. The overall

[1] https://github.com/dolthub/dolt

[2] https://www.rwth-innovation.de/en/aktuelles/aktuelle-detailseiten/bmwi-funded-project-ki-last

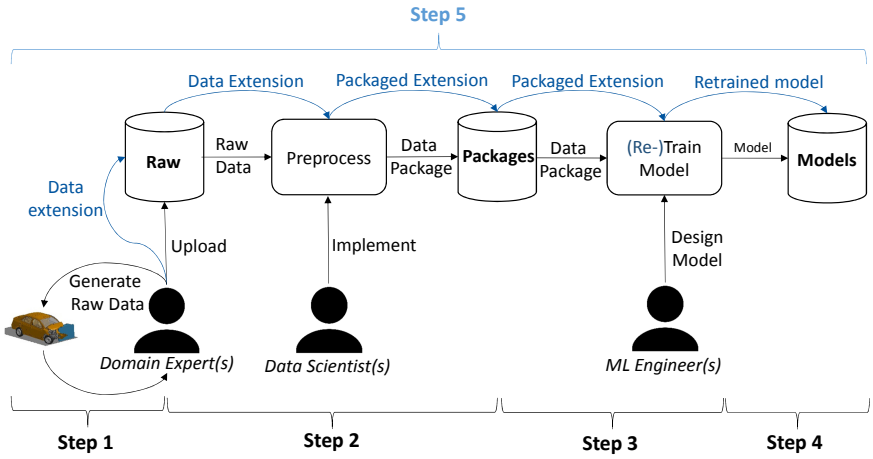

Figure 1: Abstracted Workflow Steps from our Running Example

project goal is that the AI is capable of generating lattice structures inside a given installation space conforming to the desired performance when being exposed to different load-cases. The running example in this paper is given as a first step towards the fulfillment of the overall project goal, namely an ML model intended to serve as a cheaper and faster simulation surrogate. In particular, it should use the parameterized lattice structure as the input and output of the mechanical performance, i.e. the Energy Absorption (EA) and the Peak Crush Force (PCF) [25].

4.2 Workflow

Workflow, Step 1: The workflow in the development process begins with the generation of the training data. As the ML model should surrogate the simulation, the training data is accordingly extracted from simulations to make the ML model learn patterns between the input and output. It is then intended to perform predictions for unseen data without needing to calculate the time-consuming simulation with the expensive finite element analysis. As the configuration and execution of the simulation require experience in the tooling as well as a consolidated scientific background, it is executed by different *domain experts* with appropriate facilities including a computing cluster. In our use-case, the *domain experts* use LS-DYNA[3] for executing the simulation. From the ML perspective, this process produces raw data consisting of large binary files. These files are uploaded to a cloud storage provider. The raw data remains saved unchanged in any case, as it occurs different parts of it are relevant. For example, sometimes not only the lattice parameters but also the lattice structure as a graph may be a relevant feature for a model to learn from.

Workflow, Step 2: After being uploaded to the central storage, the raw data has to be transferred into an appropriate feature-label format that can be applied to a training procedure. The features in our case are the lattice structure parameters, while the labels are given by the EA and PCF. All of this information is implicitly comprehended in the binary files produced by LS-DYNA, which are preprocessed with the help of a Python library and arithmetic calculations. The person undertaking this task assumes the role of a *data scientist*. The *data scientist* has to extract the relevant features in close consultation with the *domain experts*, as they have the physical background to tell which features may allow a machine to learn a pattern between the input and output variables, while the desired output variables are a choice of the domain experts as well. Based on the data added by the *domain experts*, the *data scientists* initiate a process, that executes the preprocessing script, packages the resulting dataset, and assigns a matching version number to the dataset before uploading. As a package, we consider an artifact in the shape of an archive containing the training data as already introduced as DAR in [4].

Workflow, Step 3: For the next step of the workflow, the *ML engineers* come into play, who develop the ML models using the deployed datasets from the previous step. The datasets conform to the target format to be fed into the network. The *ML engineer* simply has to specify the dataset, which is a clean dataset that is then installed locally and serves for pending training processes. In our running example, different ML approaches are built by different *ML engineers*, for example, one tries predicting the EA and PCF based on the lattice structures parameters, and another one works on the lattice structure interpreted as a graph trying to predict the stepwise deformation procedure. The development process itself is experimental, so different architectures and hyperparameters are tested, which is already enabled through the model-driven engineering approach of MontiAnna and the corresponding code generation based on the configuration of the model [19]. The trained model is, at this point, usually only a proof-of-concept prototype, because unnecessary compute-intensive and hence expensive simulations should be avoided, if the ML model is unable to learn from the derived data anyways.

Workflow, Step 4: After the *ML engineer* has successfully trained a model, she packages it and delivers it to a central storage[4]. Before it is uploaded, the package is tested on its functionality by being executed in a pipeline. This test prevents corrupt implementations from being deployed. *Domain experts* and *data scientists* can then access the model and test or productively use it.

[3] https://www.dynamore.de/en/products/dyna/Introduction?set_language=en

[4] We are aware, that in terms of MLOps it is good practice to deploy a complete pipeline instead of only the model (i.e. the weights) only [28]. In our case, we define the model to include the complete pipeline.

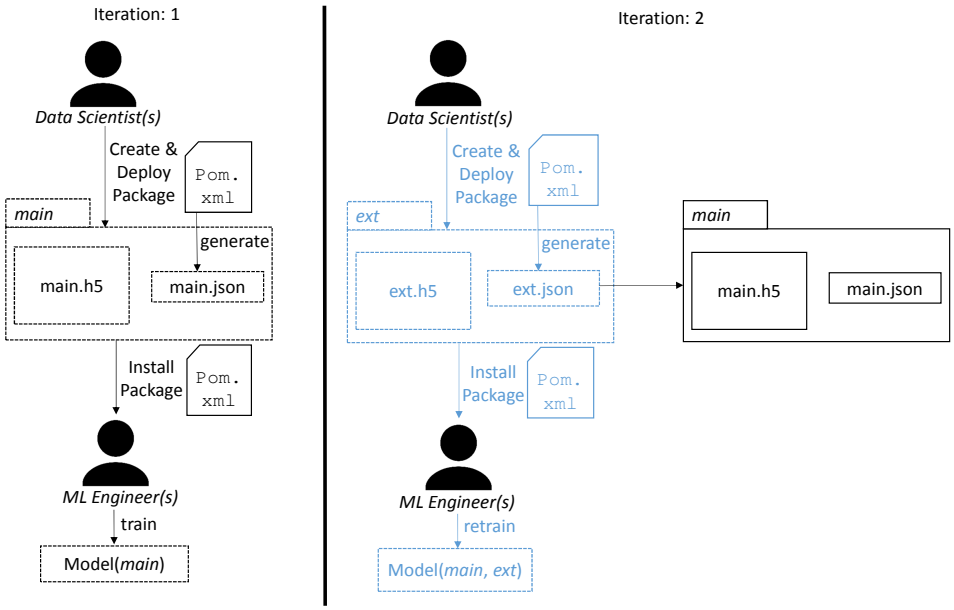

Figure 2: Concept for dealing with iteratively growing datasets

Workflow, Step 5: After the steps **1-4** are executed, the result is a deployed model. After this deployment, there is often an enlargement of the dataset generated by one or more *domain experts*. The reason for that is that the enlargement is put into the prospect of continuously improving the prediction quality of the model or being capable to perform predictions inside a new subsection of the data's distribution. Including this enlargement, we repeat steps **1-4** to create an enlargement of the dataset. Following that, this procedure is illustrated in Figure 1, where the associated steps are colored in blue. As a consequence, the raw data increases iteratively, caused by the explicit generation of more training data from multiple *domain experts*. These extensions have to be both managed from the data point of view and the functionality to use them for retraining has to be implemented. The integration of these ideas into the MontiAnna framework form the main contribution and the novel idea of this paper. For this aim, we now derive requirements to enable this procedure.

4.3 Requirements

The requirements are derived from the workflow to support it conveniently. Especially, we intended to enable **Step 5**, the retraining, which is intertwined with the previous steps, as on the one hand, enabling **Step 5** influences the concepts behind some of the previous steps, and on the other hand, it implies the automation of the previous steps.

R1: Data Management

- **R1-1 Extending Datasets:** The *data scientist* must have the possibility to mark an existing dataset as an extension of an existing dataset.
- **R1-2 Storage efficiency:** The extension should be stored in a storage efficient way. In particular, the extension should not include data from the dataset it extends.
- **R1-3 Versioning:** The version number of the dataset should be managed in accordance to the extensions to guarantee traceability.

R2: Continuous Retraining

- **R2-1 Retraining:** The *ML engineer* should have the possibility to improve models through retraining, whenever new data is generated as an extension of an existing dataset.
- **R2-2 Computational Efficiency:** To save computational power, the retraining procedure should be implemented efficiently.
- **R2-3 Continuity:** The retraining procedure of the model can be configured to be executed completely automated after the raw data was uploaded and specified as an extension by the *data scientist*.

5 DATA MANAGEMENT AND CONTINUOUS RETRAINING

We illustrate our model-driven concept for dealing with iteratively growing datasets in Figure 2. The figure can be interpreted as an internal look into steps **2** to **5**. In our following explanation, we repeatedly refer to this image.

5.1 Data Management

This subsection focuses on the upper part of Figure 2. To meet the requirements **R1: Dynamic Data Management** from the previous section, we introduce our concepts of extending datasets in a chained manner.

In Figure 2, there is one dataset in the first iteration, that has the id *main* and the data file inside has the ending *h5* which stands for a Hierarchical Data Format version 5 (HDF5) file, a file format often used in the ML context for hierarchically saving data with a tree-like structure [10] [16]. Anyways, we are not restricted to HDF5, so any other training data file type can be integrated into these

concepts as well. The contents of this file were gathered by the *data scientist*. At this point, it does not matter whether they resulted from an explicit generation of raw data, or whether they were passively generated e.g. from sensors. That is why we leave out **Step 1** in our concept description, as it is not the focus of this paper. Based on the dataset, a package is created and deployed using Apache Maven. We, therefore, developed a plugin, so that the functionality is capsuled and easily configurable through defining a `pom.xml`. At the same time, a JSON file is automatically created which stores metadata about the dataset. We decided on JSON because storing the information in the metadata part of the HDF5 file would require the use of HDF5 files, which cannot be ensured for all ML libraries. This would thereby harden the flexibility.

Now we consider the case, that the raw data is extended and the *data scientist* wants to deploy an extension of it, which has, in the case of Figure 2, the id *ext*. With the help of our framework, the *data scientist* can now refer to the existing already deployed package by defining its id and version number. Internally, this works through the JSON file that contains a reference pointing towards the base package. This functionality is also completely included in configurable Maven goals so that the configuration is solely modeled in a `pom.xml`. Through this idea, **R1-1** is met. Extensions of datasets do not necessarily have to refer to base datasets, but can also refer to an extension themselves. As a consequence, a chain of extensions can be built to continuously grow in parallel to the growth of the raw data. This means, in a potential third iteration could contain an *ext_2* that is inside a package with a JSON file pointing towards the package with the dataset *ext*. This functionality forms the dataset model.

As we are only using pointers and not saving any duplicates or the union of an extension together with the base dataset, storage efficiency from requirement **R1-2** can also be seen satisfied. The only overhead we must mention is the JSON files, but their size is negligible when compared to the size of common training datasets.

When an extension is deployed, the version number needs to be increased. In this manner, **R1-3** is satisfied. A version number is unique and can only be assigned to one dataset. Overwriting can be enforced if the version number is explicitly set and the validation is turned off. The version number proves to be very useful in terms of data lineage and reproducibility, a fundamental concept of MLOps [2]. The development process of ML models is experimental, that is why multiple experiments with different datasets can be carried out. Here, it is always guaranteed that the dataset can be both identified and reconstructed.

In our previous examinations, we solely considered the training dataset. Usually, there is always part of the data kept out for validation and testing. For reasons of clarity, these datasets are not depicted in Figure 2. Nevertheless, the *data scientist* can define a test and validation dataset in the `pom.xml`. The test and validation datasets are optional and can be contained in any package, i.e. in the base dataset or any extension. The two then work for all trained models based on new extensions, so that e.g. the performance before and after training with an extension can be compared with the same dataset. If this is not desired, they can also be redefined or altered when adding a new extension.

5.2 Continuous Retraining

After we discussed the data-management concept visualized in the upper part of Figure 2, we now focus on the lower part illustrating the integration of training and continuous retraining methods. In the first iteration, the *ML engineer* trains a model with the data-package containing the *main* dataset, that can be received by defining the package inside the `pom.xml` as a dependency. In this manner, it is automatically checked whether it is already locally available or still needs to be downloaded to be fed into the training routine, that is initiated and configured with the same `pom.xml` build file.

The key part of our work is represented by the second iteration, where the *ML engineer* is delivered an extension of the dataset she wants to retrain the model with. To receive it, she has to refer to this extension by defining its id and version as a dependency in the `pom.xml` file. The chain of datasets is then automatically resolved, so all previous datasets are automatically downloaded, if not already locally available.

Various approaches exist explaining how to reuse and retrain already trained models [21]. These mainly focus on the case, where new data is intended for the prediction of a new feature, e.g. detecting whether construction workers wear a safety vest after a model has already been trained to detect whether they wear a hard hat. In these cases, the performance of the ML models tends to massively deteriorate after retraining for the new prediction task concerning the performance of the old prediction task. However, for our current concept, we focus on data to solely improve the accuracy or to cover a new domain in the distribution of the training data. Consequently, the goal in the second iteration is to produce a model, that has the same performance as a model, that would have directly been trained on the union of the samples from the *main* and *ext* dataset, but takes less time for this. Therefore, the fine-tuning approach in [21] inspired our concept, although even the last layer remains the same before starting the retraining because no new prediction task is intended.

Accordingly, retraining is realized through initializing the ML model with weights learned from a previous training procedure and trying to improve these weights by executing a new training procedure with the new data. If the *ML engineer* references an extension and has not already trained a model based on a previous dataset, the training is performed with the union of the datasets. Through the concept described in this section, both **R2-1** and **R2-2** are satisfied, as the retraining is capable of improving already trained models in a computationally efficient way, as only the data is used for retraining, that has not already been used in a training procedure. The result, as shown in Figure 2 of the retraining procedure is a model with extracted knowledge from both the *main* and *ext* dataset.

The retraining can also be configured conforming to the model-based manner in the `ConfLang` (cf. Section 2). Several parameters influence the learning process, for example, the selected optimizer and its learning rate. An excerpt of this configuration model for the two cases is shown in Figure 3. The *ML engineer* can decide, whether she would like to continue the retraining with the parameters of the training procedure of the base dataset (Figure 3a), or whether she would like to explicitly set them for the retraining (Figure 3b). As the retraining parameter is automatically adjusted during the

```
configuration Network{
  [...]
  optimizer : sgd {
    learning_rate : 0.1
    learning_rate_decay : 0.85
  }
  retraining_type : automatically
}
```

ConfLang

(a) Retraining type is set to automatically, so the parameters from the original training process are overtaken.

```
configuration Network{
  [...]
  optimizer : sgd {
    learning_rate : 0.1
    learning_rate_decay : 0.85
  }
  retraining_type : manually
    retraining_optimizer : sgd {
      learning_rate : 0.12
      learning_rate_decay : 0.83
  }
}
```

ConfLang

(b) Retraining type is set to manually, so the configuration must be done explicitly.

Figure 3: Automatic versus manual retraining configuration

training procedure, the *ML engineer* can even continue with the learning rate to pretend one big training, or she can increase the learning rate to attach more importance to the extension.

To enable the continuity aspect, the retraining procedure needs to be automated as required by **R2-3**. By means of continuous retraining, the prediction quality can be improved automatically without the *data scientist* or *ML engineer* even interacting. In this way, the improvement caused by the iterative enlargement of datasets can be retraced by looking at the deployed model as agile as possible. For the automation, we implemented an event-based retraining pipeline that is triggered by the deployment of an extension by the *data scientist*. The following retraining and deployment of the retrained model can also be fully automated and be attached to conditions (e.g. a better accuracy than the old model), or it can be specified, that the *ML engineer* should first manually confirm the model before it gets deployed.

6 EVALUATION

We are going to evaluate the discussed data management concepts on our running example aiming to answer the following research questions.

RQ1: Training Time and Quality: Can intelligent data versioning with continuous retraining save storage resources and training time while preserving or improving the model quality?

RQ2: Agility: How can automation contribute to an agile development process for data-driven applications?

For the evaluation, we fall back on our running example introduced in Section 4. The lattice structures have a cubic shape, and both an exemplary initial structure and its deformed version can be seen in Figure 4. As an input, we use nine parameters describing the lattice structure, amongst others the beam thickness and rotation. As the output, we try to predict the EA and PCF. The architecture

Figure 4: Initial and deformed Lattice structures

	w/o retraining **A)**	w/ retraining **B)**	Δ
Avg. CPU time	41.83s	27.20s	−14.63s
MAE	7.85	6.71	−1.14

Figure 5: Time and MAE of the to setups A) and B) to compare.

consists of four fully-connected layers with ReLu activation functions and a total of 226 neurons. For our comparison, we first divide our dataset of 400 samples into a training (90%) and a testing (10%) dataset. We then compare the Mean Absoulte Error (MAE) and training time considering the workflow before our data management approach in a case we call **A)** and with the data management enabled in a case called **B)**. We divide the dataset into a base dataset (60% of training) and an extension (40% of training). For case **B)** we train first on the base dataset and then perform a retraining solely on the extension. For case **A)** we first train on the base dataset and then on the union of the base dataset and the extension, as this was previously the only opportunity to improve the model performance if the dataset update is integrated into the original dataset directly. The data in the following table shows the averaged time and MAE repeating the experiment 100 times.

Based on only our experiments for this special use-case, we can answer our **RQ1** by saying that around 40% of the training time could be saved there. In the approach without retraining, it was trained on 160% of the samples, the reduction in time of only 40% is to explain the overhead of loading the weights for retraining again. The MAE decreased by around 15%. However, we observed a high variance in MAE at the end of the training so that the difference can be considered random. Hence, we can claim that in our experiment the accuracy roughly remains the same. A more detailed analysis of the influence of retraining approach on the resulting model quality is subject of future work.

To answer **RQ2**, we pertain to our experience in the KI-LaSt project introduced in Section 4. After implementing this concept, we were able to fasten the feedback loop drastically. The *domain experts* first came up with a base dataset and expected feedback in form of a proof of concept model. Then, they generated raw data to improve the model. After the *data scientist* preprocessed the data, the complete retraining and deployment procedure was automated, so that the experts obtained feedback within the next days. Before, the *ML engineer* had to merge the existing and the new dataset by hand and train the model all over again. Even in cases, where the requirements changed, which is symbolical for agile processes, the intelligent versioning and automation sped up the development process, because only the preprocessing and the model had to be changed inside the fully automated workflow. The development of

the application, which is experimental and often requires multiple iterations, is severely accelerated, if the steps inside are automated.

Threats to Internal Validity: The experiments cover a rather small part of the hyperparameter and architectural space. The improvement of training time and loss in accuracy might deviate for different setups.

Threats to External Validity: The research work was mainly evaluated on a single project. The results might differ when transferred to projects with different organization schemes and tools. Then, the concepts presented in this work might not lead to an increase in agility or process improvement as expected.

Threats to Construct Validity: In the presented experiment the complete dataset was known from the beginning and we performed an arbitrary splitting into a base and an extension dataset. In a real scenario data might arrive in chunks of varying sizes which would possibly affect the results.

7 CONCLUSION

In this work we introduced a novel model-driven data management framework for iteratively growing datasets and integrated an automated continuous retraining routine into this. The presented concepts enable us to deal with growing datasets and to store resulting training data efficiently and were integrated into the MontiAnna framework. The requirements were abstracted from the workflow of an actual research project and satisfied by our concept evolved throughout the paper. For the evaluation, we show how training time can be saved by applying our concepts on our running example without losing model accuracy. We conclude that our approach has the potential to save development costs and enforce agility in data-driven projects.

In future work, we are going to extend our evaluation to different ML projects and deepen our analysis regarding a more extensive coverage of the hyperparameter and architectural space. Furthermore, we plan to investigate how semantic modeling can be used to further improve the consistency between data and ML models, e.g. by verifying that the given data and its labels are compatible with the chosen network architecture and its parameters. We also aim at implementing a configurable retrain methods, for example the *Learning without Forgetting* algorithm from [21], to avoid worsening the prediction of learned patterns from previous datasets, when a new task should be trained.

REFERENCES

[1] P. Agrawal et al. 2019. Data platform for machine learning. In *Proceedings of the 2019 International Conference on Management of Data*. 1803–1816.

[2] Sridhar Alla and Suman Kalyan Adari. 2021. What is mlops? In *Beginning MLOps with MLFlow*. Springer, 79–124.

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *ICSE-SEIP'19*. 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[4] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. 2021. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In *GPCE'21*. 55–68.

[5] Amine Barrak, Ellis E. Eghan, and Bram Adams. 2021. On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects. In *SANER'21*. 422–433. https://doi.org/10.1109/SANER50967.2021.00046

[6] Marouane Birjali, Abderrahim Beni-Hssane, and Mohammed Erritali. 2017. Machine learning and semantic sentiment analysis based algorithms for suicide sentiment prediction in social networks. *Procedia Computer Science* 113 (2017), 65–72.

[7] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. Data management in machine learning systems. *Synthesis Lectures on Data Management* 11, 1 (2019), 1–173.

[8] Carl Boettiger. 2018. Managing larger data on a github repository. *Journal of Open Source Software* 3, 29 (2018), 971.

[9] Robert Culkin and Sanjiv R Das. 2017. Machine learning in finance: the case of deep learning for option pricing. *Journal of Investment Management* 15, 4 (2017), 92–100.

[10] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. 36–47.

[11] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. 2019. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In *Proceedings of MODELS 2019. Workshop MDE Intelligence* (Munich), Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel (Eds.). 196–202. http://www.se-rwth.de/publications/Modeling-Deep-Reinforcement-Learning-based-Architectures-for-Cyber-Physical-Systems.pdf

[12] Lei Gu and Huan Li. 2013. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, 721–727.

[13] Robert Ilijason. 2020. Getting Data into Databricks. In *Beginning Apache Spark Using Azure Databricks*. Springer, 51–73.

[14] et al. Jain, A. 2020. Overview and importance of data quality for machine learning tasks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3561–3562.

[15] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. 2019. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology* 18, 2 (July 2019), 1–20. https://doi.org/10.5381/jot.2019.18.2.a2 The 15th European Conference on Modelling Foundations and Applications.

[16] Sandeep Koranne. 2011. Hierarchical data format 5: HDF5. In *Handbook of open source tools*. Springer, 191–200.

[17] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. 2010. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)* 12, 5 (September 2010), 353–372.

[18] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. 2019. Modeling and Training of Neural Processing Systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 283–293. https://doi.org/10.1109/MODELS.2019.00012

[19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. 2019. Modeling and Training of Neural Processing Systems. In *MODELS'19* (Munich). IEEE, 283–293.

[20] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. 2018. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *MODELS'18* (Copenhagen). ACM, 447 – 457.

[21] Zhizhong Li and Derek Hoiem. 2017. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence* 40, 12 (2017), 2935–2947.

[22] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2018), 2346–2363.

[23] Gunasekaran Manogaran and Daphne Lopez. 2017. A survey of big data architectures and machine learning algorithms in healthcare. *International Journal of Biomedical Engineering and Technology* 25, 2-4 (2017), 182–211.

[24] Jyoti Nandimath, Ekata Banerjee, Ankur Patil, Pratima Kakade, Saumitra Vaidya, and Divyansh Chaturvedi. 2013. Big data analysis using Apache Hadoop. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE, 700–703.

[25] S. Pirmohammad and S Esmaeili Marzdashti. 2018. Crashworthiness optimization of combined straight-tapered tubes using genetic algorithm and neural networks. *Thin-Walled Structures* 127 (2018), 318–332.

[26] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1723–1726.

[27] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data lifecycle challenges in production machine learning: a survey. *ACM SIGMOD Record* 47, 2 (2018), 17–28.

[28] Philipp Ruf, Manav Madan, Christoph Reich, and Djaffar Ould-Abdeslam. 2021. Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools. *Applied Sciences* 11, 19 (2021). https://doi.org/10.3390/app11198861

[29] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.