Machine Learning (2021) 110:2993–3013
https://doi.org/10.1007/s10994-021-06052-0

MLife: a lite framework for machine learning lifecycle initialization

Cong Yang[1] · Wenfeng Wang[1] · Yunhui Zhang[1] · Zhikai Zhang[1] · Lina Shen[1] · Yipeng Li[2] · John See[3]

Received: 29 December 2020 / Revised: 14 May 2021 / Accepted: 27 August 2021 /
Published online: 14 October 2021
© The Author(s), under exclusive licence to Springer Science+Business Media LLC, part of Springer Nature 2021

Abstract

Machine learning (ML) lifecycle is a cyclic process to build an efficient ML system. Though a lot of commercial and community (non-commercial) frameworks have been proposed to streamline the major stages in the ML lifecycle, they are normally overqualified and insufficient for an ML system in its nascent phase. Driven by real-world experience in building and maintaining ML systems, we find that it is more efficient to initialize the major stages of ML lifecycle first for trial and error, followed by the extension of specific stages to acclimatize towards more complex scenarios. For this, we introduce a simple yet flexible framework, MLife, for fast ML lifecycle initialization. This is built on the fact that data flow in MLife is in a closed loop driven by bad cases, especially those which impact ML model performance the most but also provide the most value for further ML model development—a key factor towards enabling enterprises to fast track their ML capabilities. Better yet, MLife is also flexible enough to be easily extensible to more complex scenarios for future maintenance. For this, we introduce two real-world use cases to demonstrate that MLife is particularly suitable for ML systems in their early phases.

Keywords Machine learning · Machine learning lifecycle · Machine learning system · Deep learning · Data flow

1 Introduction

A significant difference between academic research and industrial development in machine learning (ML) is the *stability* of data. In academia, data is relatively stable, especially for domains where standard datasets have been established, such as ImageNet (Deng et al.

Editors: João Gama, Alípio Jorge, Salvador García.

✉ Yipeng Li
yipeng.li@outlook.com

[1] Horizon Robotics, Nanjing, China

[2] Clobotics, Seattle, USA

[3] Heriot-Watt University Malaysia, Putrajaya, Malaysia

2009) for image classification. The goal then pivots towards the quest for better algorithms to improve the performance of ML models on these established datasets. More often than not, there are no datasets readily available for ML model development in industrial applications. Even with datasets to start with, they may not be representative of real life scenarios since many *badcases* (a phrase which collectively denotes failure cases, rare cases and long tail cases in literature Bengio 2015) can only be captured in the late "using" phase (Zaharia et al. 2018). While algorithms in industrial applications are relatively stable after the delivery phase, ML model development is mostly enabled through the accumulated variety of data driven by badcases (Sculley et al. 2015). In other words, ML models are iteratively updated using data collected from unseen scenarios to ensure the robustness and stability of the ML system. For example, Fig. 1 presents the normalized number of reported badcases in two real-world ML systems (Clobotics 2021; Horizon Robotics 2021). We found that most cases were solved in the first 4–6 weeks while the rest were constantly returned from customers. In practice, *ML lifecycle* is seen as an important mechanism to clarify the stages involved in each ML model iteration. Following the convention in Ashmore et al. (2019), Miao et al. (2017), ML lifecycle is a cyclic process to build an efficient ML system.

Figure 2 illustrates the five major stages in a complete ML lifecycle. Two important characteristics can be observed: (1) ML model iteration is driven by the existence of badcases in data flow. (2) All five stages have high intrinsic complexity, and are dependent on other stages. As a result, the stability of an ML system is highly dependent on the efficiency of data processing and collaboration between these stages. However, considering the complications and coverage, we find that both commercial (Datatron 2021; Peltarion 2021; Algorithmia 2021; 5Analytics 2021) and community (i.e. non-commercial) (kubeflow 2021; airflow 2021; Miao et al. 2017) solutions are sub-optimal for initializing an ML system, particularly in its early phases. This is because commercial frameworks are normally overqualified and expensive for an ML system which requires trial-and-error procedure to fix its business-facing (i.e. live) model. Moreover, community solutions are hard to be applied directly as they either do not streamline all stages in the ML lifecycle, or have limited ability on data and badcases management. The instability of data which contributes to these badcases, and the inability to make the most out of them is the greatest limiting factor for companies, particularly startups, to initiate their ML capabilities as well as to successfully build industrial applications (Sculley et al. 2015).

In this paper, we introduce a simple yet efficient framework, *MLife*, for fast and effective initialization of the major stages of ML lifecycle. Particularly, it contains a set of data management tools especially catered for badcase management, which can effectively guild ML

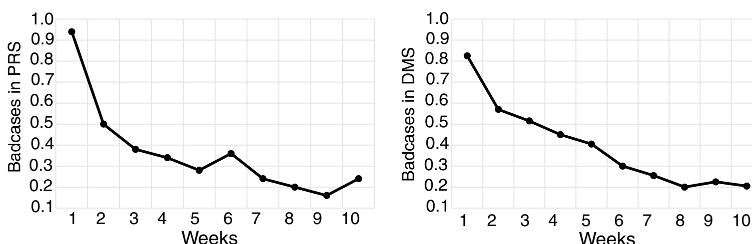


Fig. 1 Normalized number of reported badcases along with the weekly ML model iteration in two real-world systems: product recognition system (PRS Clobotics (2021), left) and driver monitoring system (DMS Horizon Robotics (2021), right)

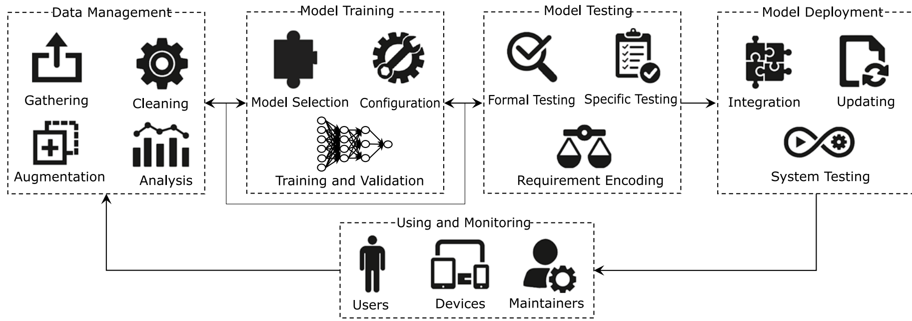


Fig. 2 A typical ML lifecycle in practice (Ashmore et al. 2019; Clobotics 2021; Horizon Robotics 2021). In the first step, *Data Management* mainly focuses on gathering and processing data for training and testing ML models. For gathering, some tools for data uploading and labeling are normally required (Russell et al. 2008) while in processing, tasks ranged from data analysis (Liu et al. 2017) to preparation and augmentation (Lee et al. 2019). *Model Training* is the actual machine learning stage whereby pre-designed ML models are properly selected, configured, trained and validated via an inner loop until a promising model is obtained (Amazon 2020). *Model Testing* is responsible for checking the performance and limitations of the trained model. By comparing with requirements, the model can either be deployed or dropped based on the results. In *Model Deployment*, the trained and tested models are integrated with other models, or are used to replace their predecessors in the ML system. In this stage, an end-to-end system testing is typically required. Once the first 4 stages are finished, the ML system is released to users in the *Using and Monitoring* Stage whereby the operational data from sensors, user feedback and execution results are periodically send to maintainers for further steps

model development for industrial applications. MLife is built on the assumption that the components in Fig. 2 can be reorganized into a set of hierarchical layers, as depicted in different concentric circles shown in Fig. 3: (1) The middle layer (line patterned background) is a set of simplified and standardized operations for modeling and initializing each stage within the ML lifecycle. The main purpose of this layer is to provide crucial and general tools so that all stages can be quickly activated. All tools in this layer are fully furnished with web-based user interface thereby only negligible configuration works are required for adaptation. (2) The outer layer (gray background) is more flexible as it contains various APIs in each stage to support future extension on workflows and scenarios. (3) The inner layer (white background) is a set of generic tools to support the middle and outer layers. Tools in this layer can be updated to ensure the scalability of ML lifecycle on data volume and user numbers.

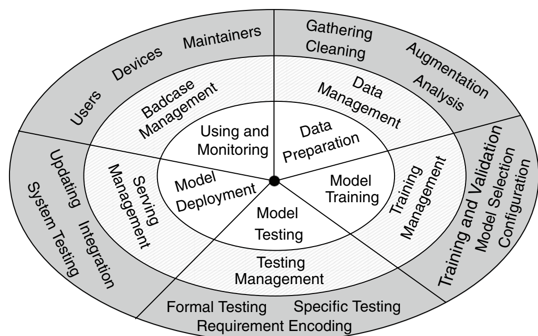


Fig. 3 Motivation of MLife: the ML lifecycle can be reorganized into the hierarchical layers

Building upon these tools and APIs in three layers, MLife is advantageous in two aspects. (1) *Lite:* A primary benefit of the middle layer is that users can quickly initialize an ML lifecycle without overloading the development and adaptation works. This is particularly meaningful for a ML system in early phases since MLife can effectively simplify the intrinsic complexity of each stage and reduce over-collaboration between different teams. (2) *Flexible:* Both inner and outer layers are flexible enough such that they can be extended to meet different requirements in terms of complexity and scalability. The rationale behind MLife is that we do not intent to build an end-to-end ML framework in the middle layer to cover the plurality of use cases. Instead, we explore and focus on the requirements with stability and consistency in different stages. This is because production ML is still in flux as developments in this field changes very rapidly, such as ML libraries, ML models, the large number of startups and their commercial fields (Polyzotis et al. 2018). Therefore, on one hand, MLife tries to standardize and enhance the tooling for individual tasks in each stage, while it also attempts to keep an appropriate level of flexibility to support the data workflow within the ML lifecycle.

The main contribution of this article is in the introduction of a framework, MLife, to facilitate quick and effective initialization of an ML lifecycle. Also, its flexibility in supporting future extensions after the initialization renders it useful for maintenance of the ML lifecycle. To evaluate its usability, two real use cases were introduced to show the efficacy and suitability of MLife particularly in the early phases of ML systems.

2 Related works

In this section, we systematically review existing tools and frameworks for the ML lifecycle. For a more detailed treatment of this topic in general, the recent compilations by Ashmore et al. (2019) and Polyzotis et al. (2018) also offer sufficiently good insights.

2.1 Tools in stages

Real-world ML applications normally require feature generation from multiple input sources, the building of ensembles from different models, and they frequently target hard-to-optimize business metrics (Schelter et al. 2018). In such cases, many of these challenges necessitate the need for practical and open-source tools that address different aspects within the ML lifecycle stages. For instance, Chen et al. proposed BigGorilla for data preparation and integration (Chen et al. 2018) during the data management stage. Particularly, BigGorilla provides a set of packages for information extraction, schema matching, and entity matching. Vartak and Madden introduced MODELDB for model management (Vartak and Madden 2018) during the model training stage. Unlike the commercial model management system deployed in the SAS platform (SAS 2021), MODELDB provides various libraries to track and retrieve provenance information of a model during the experimentation phase. In the model deployment stage, most existing tools (Microsoft machine learning server 2021; Mxnet model server (mms) 2021; Crankshaw et al. 2017; Olston et al. 2017) aim to minimize the burden of deployment by reusing the interfaces in model training and testing stages. For example, Interlandi et al. proposed PRETZEL, a high-performance prediction serving system (Lee et al. 2018a). Kaiser et al. released a serving infrastructure, Cortex, for deploying model locally and on web services such as AWS (2021). Both approaches are fully engaged with the interfaces for model training and prediction. Meanwhile, in the

using and monitoring stage, several works (Lee et al. 2019; Fan and Li 2018) have been proposed to better involve human-in-the-loop (e.g. observation and supervision) thereby to prepare and wrangle data more efficiently in the following iteration of stages. Though the aforementioned tools can improve the efficiency of specific tasks in specific stages, they cannot streamline the entire ML lifecycle due to their independent nature. Moreover, it is also challenging to integrate them in practice because of their duplicated functions and diverse requirements for setting up. Different from the 'independent' tools above, MLife not only provides a set of built-in tools in the middle layer for fast initialization, but also can be extended via the API and existing tools in the outer and inner layers, respectively. With this advantage, MLife is convenient and flexible enough to elicit different requirements from users.

2.2 Frameworks

Table 1 lists some existing frameworks and their covered stages. We find that most commercial and internal works have powerful tools to cover all stages of the ML lifecycle.

Table 1 Comparison between different frameworks and platforms for managing the ML Lifecycle

Frameworks	Usage	Init	DM	TR	TS	DE	UM
Datatron (Datatron 2021)	Commercial	√	√	√	√	√	√
Peltarion (Peltarion 2021)	Commercial	√	√	√	√	√	√
SELDON (Seldon 2021)	Commercial	√	√	√	√	√	√
Algorithmia (Algorithmia 2021)	Commercial	√	√	√	√	√	√
5 Analytics (5Analytics 2021)	Commercial	√	√	√	√	√	√
craft ai (craft ai 2021)	Commercial	√	√	√	√	√	√
KNIME (KNIME 2021)	Commercial	√	√	√	√	√	√
valohai (valohai 2021)	Commercial	√	√	√	√	√	√
AML (Microsoft 2021)	Commercial	√	√	√	√	√	√
SageMaker (SageMaker 2021)	Commercial	√	√	√	√	√	√
Tfx Baylor et al. (2017)	Internal	–	–	√	√	√	√
FBLearner (FBLearner 2021)	Internal	–	√	√	√	√	√
Michelangelo (Michelangelo 2021)	Internal	–	√	√	√	√	√
kubeflow (kubeflow 2021)	Community	△	△	△	△	△	△
Airflow (airflow 2021)	Community	△	×	△	△	△	×
Flyte (Flyte 2021)	Community	△	×	×	×	△	△
NiFi (NiFi 2021)	Community	△	△	×	×	×	△
MLflow Zaharia et al. (2018)	Community	×	△	△	△	△	×
Cortex (Cortex 2021)	Community	△	×	×	×	√	×
JupyterHub (JupyterHub 2021)	Community	△	×	△	△	×	×
ModelHub Miao et al. (2017)	Community	–	√	√	×	×	×
MLife	**Community**	√	√	△	△	△	√

Init System setup and initialization, *DM* Data Management, *TR* Model Traning, *TS* Model Testing, *DE* Model Deployment, *UM* Using and Monitoring

(need additional works[△]; support[√]; nonsupport[×]; unclear[−])

In this table, △ denotes that a framework or API cannot be used for practical deployment before considerable works on adaptation such as reconstruction, coding and compiling, etc. are added. As most commercial and internal frameworks are deployed in cloud platforms, they can be easily configured and employed via web interface. The community frameworks, in contrast, are quite disorganized to some extent. We can find that model development stages such as training and evaluation have received tremendous attention while using and monitoring is barely covered. For instance, although MLflow (mlflow 2021; Zaharia et al. 2018) contains various tools and APIs for tracking experiments, packaging ML codes, managing and deploying ML models, there is no components specially designed for data and badcase management. As a result, ML model development is loosely guided by badcases in the data workflow. In fact, 4 complications are normally addressed to evaluate the usability of a framework (Baylor et al. 2017):

- *One for many*: A framework should be generic enough to handle use cases from different domains.
- *Continuous training and serving*: A framework should support model iteration not only over fixed data, but also evolving data.
- *Human-in-the-loop*: A framework should be easy to install and configure. Besides, it should provide simple user interfaces and pipelines for maintainers and engineers at different stages of the ML lifecycle. Furthermore, it should provide minimal configuration for users to quickly check their data and models.
- *Production-level reliability and scalability*: A framework should be robust against disruptions from data, configuration and environment. Furthermore, it also should scale gracefully to the high data volume that is common in training, testing and monitoring.

Considering these complications, we find that MLife has higher usability compared to most other community solutions. Particularly, as MLife contains three layers to support different levels of requirements, users can easily initialize and extend the ML lifecycle based on even the middle and outer layers. Moreover, since MLife contains simple yet efficient tools for data management, especially in handling badcases which have a significant impact on model performance, MLife presents an essential solution to enable companies to fast track their ML lifecycle.

3 MLIFE

To simplify the stages of ML lifecycle, some tools in Fig. 3 are integrated based on their intrinsic coherence. For instance, trained ML models are normally registered and uploaded to a temporary location for testing. For each model, testing results are also attached for future management and serving reference. Thus, Model Training and Model Testing have a high intrinsic coherence and are integrated into a single component to reduce the system's complexity. As a result, as presented in Fig. 4, there are four major components in MLife: Badcase Analysis & Management (BAM), Data Analysis & Management (DAM), Model Training & Testing (MTT) and Model Serving & Management (MSM). To support the major components, a data storage system (including file and database systems) should be provided by default. In this section, we detail the tools and motivations of the middle layer in MLife followed by the overview of functions in the outer and inner layers. For better

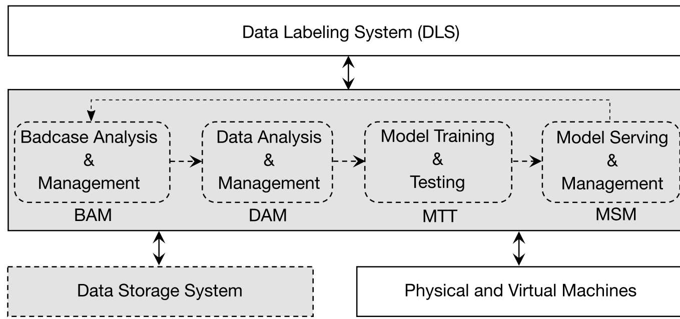


Fig. 4 Major components of MLife (the gray blocks) and its connection with other systems. (Color figure online)

description, we use a simple but typical example, expression recognition (Li and Deng 2020), to clarify the usage of components (see Fig. 5). Expression recognition is one of the key functions in driver monitoring system (DMS). The recognition results (happy, sad, angry and fear) is used for driving safety analysis, multimedia recommendation and interaction with video games, etc. In practice, the expression recognition model is iteratively updated via BAM, DAM, MTT and MSM so as to meet system requirements on accuracy and stability.

3.1 The middle layer

To streamline the major components in the middle layer, the collected badcases are analysed in BAM so as to determine the appropriate strategies for data collection and ML model training (or testing) in DAM and MTT, respectively.

BAM A badcase contains at least three pieces of information: the original data (e.g. links of image, video and audio, etc.), recognition results and ground truth. In addition to that, other attributes and operations are also necessary to collaborate with other stages. For instance, as shown in Fig. 6, ML System Functions and ML System Version indicate different gradings of badcases, e.g. system level and model level. Frequency is one of the important factors to determine the priority of a badcase. Description of a badcase should be formatted in such a way to avoid duplicated items. Besides attributes, there are also

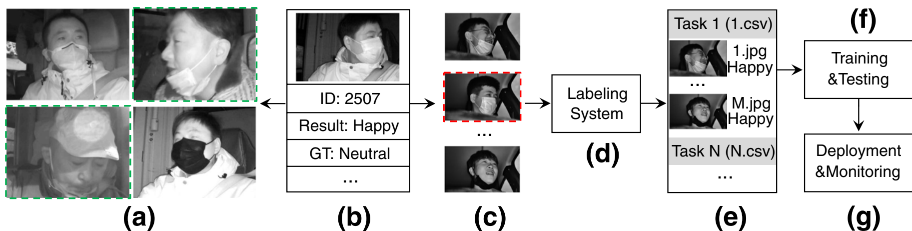


Fig. 5 Major steps of an iteration of the expression recognition model. **a** Badcase analysis. Cases of half-wearing-mask (marked in green) are valuable for expression recognition. **b** Badcase management in **BAM**. **c** Data collection and cleaning. The full-wearing-mask images (marked in red) are removed. **d** Data labelling system. **e** Data management in **DAM**. **f** Model training and testing in **MTT**. **g** Model serving and monitoring in **MSM**

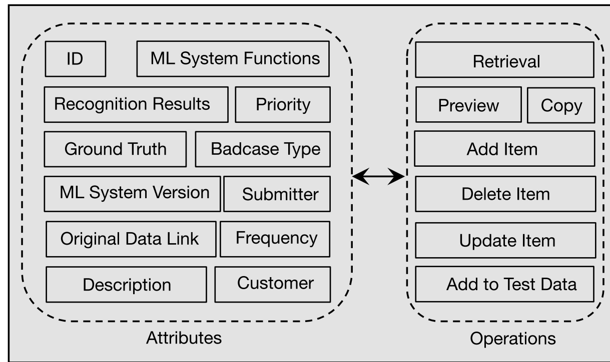


Fig. 6 Badcase attributes and the operations of BAM

some operations to improve the efficiency in badcase management. Among the operations, Retrieval is meaningful for different teams to avoid duplicated items as well as to clarify the targets in the next iteration. Add to Test Data is a process to add the badcase-related data into testing sets in DAM. It can maintain the richness of testing data thereby improving the effectiveness of model testing in MTT.

DAM Figure 7 presents the general idea of DAM. In the left part, File Data is used to store raw data that are collected with the guidance of badcases. For this, the Data Storage System (the left bottom block in Fig. 4) is used as a data store for different types of files and their meta data. After labelling, the MD5, path and annotation of files are stored in different documents with data exchange format, such as CSV (Comma-Separated Values), JSON (JavaScript Object Notation) and XML (Extensible Markup Language), etc. MD5 is used as a checksum to avoid duplicated files during the file uploading process. Data path links the raw data and its corresponding annotations. Depending on different scenarios, additional columns such as file format, file type, collection place, etc. could be added. As the data collection and labelling tasks are flexible and normally conducted with groups, or what we call *tasks* (see Fig. 7(middle)), it is more efficient to operate these tasks using the documents. Before labelling, the annotations in the documents are filled with pre-labelled data using ML models). Users can conveniently convert the documents into different format

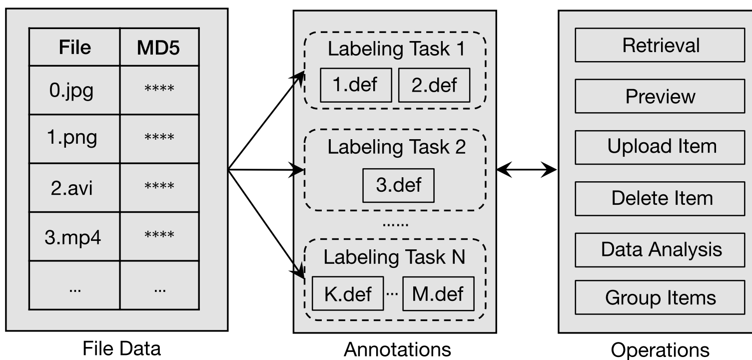


Fig. 7 General idea of DAM. *.def means the documents with data exchange format (e.g. CSV, JSON, XML, etc.) for easy handling and preservation of file locations and their annotations

to meet the requirements of labelling systems. With annotations in the documents, the Data Analysis and Group Items operations are dynamically used to check data distribution of selected documents as well as to integrate them into training and testing sets for MTT (see Fig. 7(right)), respectively. Particularly, users can dynamically analyse the data distribution in terms of classes, types and sizes using Retrieval, Selection and Data Analysis operations thereby determining additional data collection and annotation strategies. Such analysis is extremely important to ensure the quality of training and testing data is maintained at a consistent level. This is because highly balanced data with a small domain gap is one of the key contributing factors towards ML model performance (Deng et al. 2009). Specifically, these grouped documents are used for ML model training and testing. Since each document contains both file path and annotation, users can directly parse through them to download and pack the file data with labels suited for different training frameworks.

MTT The general idea and workflow of MTT is presented in Fig. 8. In the preparation step, once the documents are grouped into training or testing set using the Group Items operation in DAM, a Task ID is automatically generated and assigned. This ID is important to link all the relevant data/metadata of training and testing tasks including the documents, ML models, training logs and testing results. For the testing tasks, the Model ID should be selected in the preparation step, unlike that for the training tasks. This is understandable as the testing set and the target ML model should be well clarified before the testing process. Once the training and testing tasks are configured and initialized, the corresponding training and testing codes can be triggered. As shown in Fig. 8 (middle), we encourage users to organize their codes under three parts: (1) Training: For ML model training and validation. Once the training process is finished, the trained ML model is automatically uploaded to BAM. In addition, the training logs and configuration files will be stored in the Data Storage System. (2) Testing: For ML model testing. Similar to training, the testing results and related configuration files are stored in the Data Storage System. (3) Common: the shared libraries.

It should be noted that we do not intend to provide the Training, Testing and Common codes in MTT since different users may use different frameworks, ML models and testing strategies. To ensure the usability of MLife, a set of APIs can be provided in MTT for the purpose of uploading ML models, logs and results. Moreover, the final results of training and testing could be easily uploaded, accessed, retrieved via a web-based user interface (Fig. 8(right)). To better understand the testing results, users can also add badcases using the hyperlink between BAM and MTT. Those badcases will be used as the references of each ML model during the deployment process in MSM. Thus, users can either adapt their

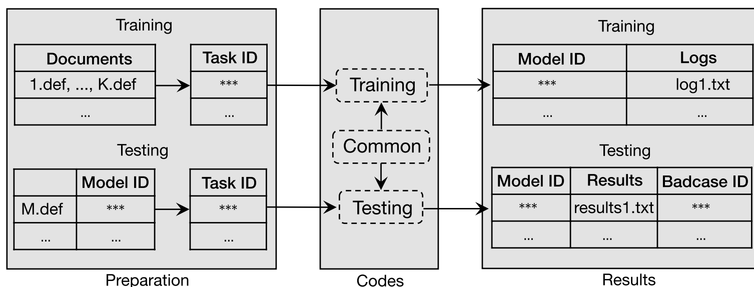


Fig. 8 General idea and workflow of MTT. For each training and testing task, a Task ID is assigned for the purpose of tracking and retrieval

training and testing codes using the proposed APIs, or directly manage the configurations and final results using the operations via the user interface. Either way, the training and testing workflow cans be easily initialized in MTT.

MSM As shown in Fig. 9, MSM contains two major parts: ML model management and ML model serving. Here, each uploaded ML model is assigned with a Model ID for indexing and connection with MTT. The Model Management part aims to store, annotate, discover, revert, and manage models in a central repository. If a trained ML model has passed the testing, users can trigger the model serving process using the Deployment operation (Fig. 9(middle)). MSM also has an operation that links with BAM for the purpose of badcase management. In practice, this operation is useful as there might be some badcases that were returned from testing in MTT or were based on feedback from customers, for before or after serving, respectively. Thus, the Badcase ID list in BAM is actively updated and attached in each deployment so as to identify the weaknesses for the benefit of subsequent iterations of ML models.

In the middle layer of MLife, Model serving is a standardized workflow. Firstly, a Deployment Task ID is created once the deployment operation is triggered. After that, users should select the target ML models (multiple or single) to be deployed. With this step, the Model IDs and their corresponding Deployment Task ID are linked. In addition to the existing badcases of each ML model from MTT, users can add more badcases based on feedback and observations from various resources. Since different metrics (e.g. Client ID, Device ID, Pending Time, etc.) may be needed in different scenarios, MSM should also be sufficiently flexible to modify the metric list. Similar to MTT, the middle layer of MSM mainly focuses on workflow initialization. The deployment action is performed via the extended APIs in the inner and outer layers. This is because the deployment techniques are different on cloud-based and edge-based ML systems (Lee et al. 2018b) while their model management and serving workflows are similar. Thus, the proposed multi-layer MSM is flexible and general enough to initialize workflows in ML systems.

Based on attributes and operations in the four components, efficiency of human-in-the-loop can be standardized and improved. This is built on the fact that ML lifecycle is essentially a process to transfer knowledges from human into machine (ML models) to solve similar and repetitive problems. The transferring process normally include data labelling as well as different operations in each iteration. For instance, most badcases regarding expression recognition were the drivers with masks during the pandemic of COVID-19 (Fanelli and Piazza 2020). To timely alarm the instability, as shown in Fig. 5a, b, operations like Retrieval, Statistics and Preview can be employed to explore the major reasons and

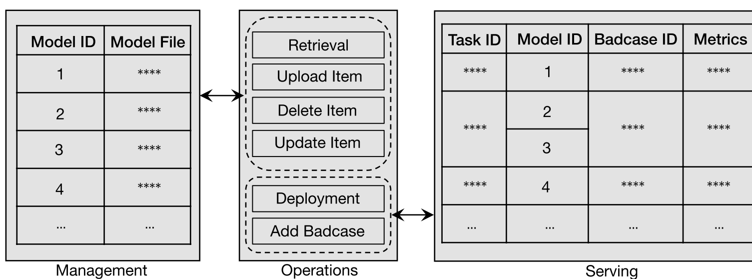


Fig. 9 General idea for model management and serving in MSM. Each serving task contains multiple fixed and user-defined metrics

solutions. Based on analysis, we found that collecting data of half-wearing-mask (wearing a mask around/below the chin) has the most significant effect. This is because robust expression recognition normally require eye and month regions. Moreover, Data Analysis operation in DAM indicated that most drivers intend to drop-down their masks below the chin when they drive alone. Thus, removing full-wearing-mask images from the training data can potentially improve model generalization and system stability (see Fig. 5c–e). After training, operations like Add to Test Data in BAM and Group Items in DAM were used to create testing data with different type of masks to validate the effectiveness of the trained model in MSM. The detailed improvements on efficiency is reported in Sect. 5.2.

3.2 The inner and outer layers

Figure 10 presents the details of inner and outer layers (as illustrated earlier in Fig. 4). We can find that the outer layer is more flexible for maintaining tasks while the inner layer is more general for supporting different tasks and tools in the middle and outer layers. The rational behind is that APIs in the outer layer are designed to adapt different domains and operations, ranging from data processing to visualization, target classification to detection and edge- to cloud-based systems. In contrast, APIs in the inner layer are mainly designed to adapt various frameworks and different volume of data and users. In some cases, APIs in both outer and inner layers are involved. For instance, automatic training requires not only data preparation, model training, testing and management in the outer layer, but also work flow management and monitoring in the inner layer.

Specifically, the inner layer contains a set of tools and classes comprising of three major components: (1) Data Storage System, which stores training and testing data (e.g. `0.jpg` and `1.def` in Fig. 7), ML models and different types of result files (e.g. `log1.txt` and `results1.txt` in Fig. 8). It should be noted that databases (e.g. SQL and NoSQL) are employed for storing annotations in some startups. For this, the documents with data exchange format can be easily imported and exported using the built-in tools in databases. (2) Deep Learning Framework, which is meant for supporting a variety of popular frameworks including Tensorflow (Tensorflow serving 2021), Pytorch (Pytorch 2021) and mxnet (mxnet 2021). (3) MLflow (Zaharia et al. 2018), which is for recording and tracking the metrics in training, testing and serving tasks. (4) AirFlow (airflow 2021), which is for scheduling and running complex pipelines. It can ensure each task of a pipeline will

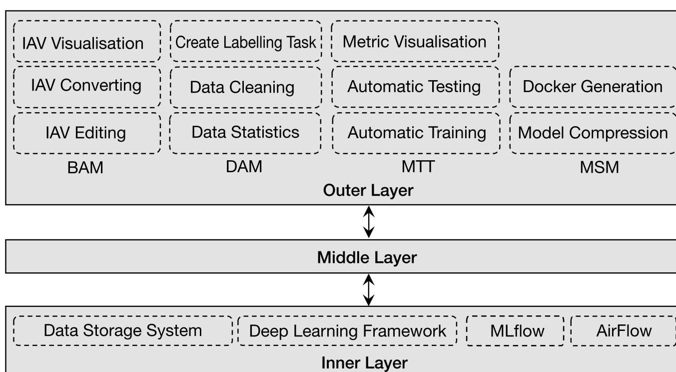


Fig. 10 APIs and tools in the inner and outer layers. IAV: image (I), audio (A), video (V)

get executed in the correct order and each task gets the required resources. The rationale behind providing these components is to ensure the scalability of MLife in terms of data, framework and workflow.

To ensure the flexibility of the outer layer, each component of MLife contains different APIs for specific purposes: (1) In BAM, APIs are mainly used for data (image, audio and video) editing, conversion and visualization to support various badcase forms and to provide detailed analysis. For instance, users may localize segments within a video that the badcase had occurred by extending APIs in IAV Editing. By extending APIs in IAV Converting, the badcase data could be converted to different formats prior to adding them to the testing set. (2) APIs in DAM are mostly focusing on data processing and statistics. For example, APIs in Data Cleaning can be extended to clean duplicated and unqualified data. APIs in Data Statistics can be used to analyze the distribution of data and the domain gap for different tasks like classification, segmentation, regression and detection, etc. Meanwhile, APIs in Create Labelling Tasks can be extended to connect with other data labelling tools for fast creation of annotations. (3) In MTT, APIs can be extended to improve the automation of training and testing tasks. For instance, Airflow (airflow 2021) can be used to incorporate an end-to-end testing pipeline by encapsulating and connecting data processing approaches and ML models. Similar to BAM and DAM, a set of APIs are also provided for data visualisation. Moreover, they can be extended to support different data formats and layout forms. (4) Different from other components, the APIs in MSM only focus on Docker Generation and Model Compression. This is motivated by the fact that existing ML models are mostly deployed in clouds and edges (Pan and McElhannon 2018). For this, docker generation and model compression are commonly required functions in cloud- and edge-based ML systems, respectively. In addition, the above APIs can be used to work with existing popular ML frameworks. For example, one may use the APIs in MTT and MSM to call Kubeflow (kubeflow 2021) pipelines, which in turn may include operations that employ Tensorflow (Tensorflow serving 2021) framework for model training, testing and serving. If a user employs public cloud services [e.g. AWS sagemaker (SageMaker 2021), Azure (Microsoft 2021)] for model training, testing and management, the APIs (and operations) in BAM and DAM can be used to manage badcases and data thereby improving the efficiency of model iteration. On the whole, MLife can be used for not only ML lifecycle initialization, but also for various maintenance and subtasks in the ML lifecycle.

4 Implementation

MLife is developed using Flask (Aslam et al. 2015) which is a micro web service framework written in Python. This is meaningful in practice since widely used deep learning and machine learning libraries are all available on the Python ecosystem (Raschka and Mirjalili 2019). Hence, this encourages users to integrate and extend MLife within their ML systems. In addition, Flask supports extensions that can add application features as if they were implemented in Flask itself. Such a feature is inherently consistent with the motivations behind the design of inner and outer layers in MLife. In this part, we discuss the details of the project structure and user interface of MLife. We intend to make MLife publicly available to the developer and research community.

The infrastructure of MLife is presented in Fig. 11: (1) The top block is the web-based user interface. Some basic operations such as checking null values and filtering are carried out in this part using Python and JavaScript. (2) The middle block is a set of tools and APIs

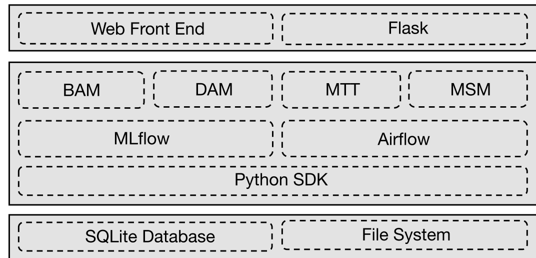


Fig. 11 The infrastructure of MLife is composed of three major parts: Web-based user interface (top), tools and APIs (middle) as well as data storage and supporting tools (bottom)

to form different layers of MLife. Particularly, APIs in the outer layer and operations in the middle layer are developed based on Python SDK. Tools like MLflow (Zaharia et al. 2018) and Airflow (Zaharia et al. 2018) are employed for the purpose of monitoring and encapsulating different tasks, respectively. This is based on the fact that there are various tasks and sub-tasks within BAM, DAM, MTT and MSM. As such, it is more flexible to encapsulate and connect them using Airflow. Meanwhile, MLflow is mainly used to monitor the training and testing tasks in MTT when users prepare the codes. The monitored data can be shared and visualized via the hyperlink between MLife and MLflow. (3) The bottom block is the database, file folders and monitoring tools. For instance, we monitor each folder to control the maximum number of files. We also synchronize file paths, MD5 and other metrics with the database using the monitoring tools. It is worth to clarify the difference between MLife and MLflow. MLflow aims to record and track metrics in model training, testing and serving tasks. In other words, MLflow is only correlated to some operations in **MTT** and **MSM**. Differently, MLife covers all steps within the ML lifecycle. Particularly, it provides various tools and APIs to improve the efficiency of badcase and data management in **BAM** and **DAM**. Besides, MLife provides APIs to encapsulate data collection and processing, model training, testing and serving steps to improve workflow automation based on AirFlow.

Figure 12 presents screen shots of MLife regarding badcase management (left) and statistics (right). For a single case, more details can be observed after clicking its preview image. For multiple retrieved and selected cases, automatic statistics can be applied to determine priorities in the next round iteration. For instance, we can clearly find that more than half (57.1%) of the retrieved badcases are related to emotion (expression recognition). Similarly, Fig. 13 presents the screen shots of data management (left) and statistics (right). We can use keywords or Advanced Search to retrieve documents for further statistics on data quality and quantity. It should be mentioned that Airflow and MLflow are not embedded in the middle layer of MLife but merely used to supplement MLife. In other words, the middle layer is purely built on Python SDK and only contains Flask-based APIs and web pages, SQLite database (Bhosale

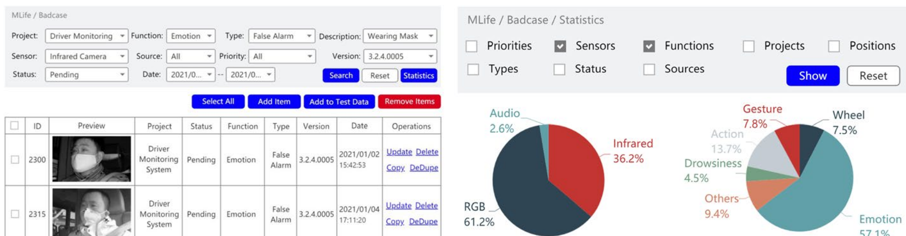


Fig. 12 Screen shots of MLife: Badcase management and retrieval (left) and statistics (right)

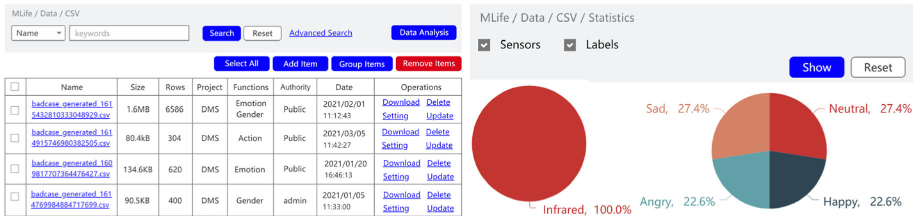


Fig. 13 Screen shots of MLife: Data management (left) and statistics (right)

et al. 2015) and folders for file storage. This can maximized the lightness and efficiency of MLife for fast ML lifecycle initialization since additional adaptation is not required in the middle layer. For example, SQLite is a self-contained, serverless, zero-configuration and transactional SQL database engine. Different from most other SQL databases, SQLite does not have a separate server process. The benefit of this is that once the required packages are installed in Python, the middle layer of MLife can be activated right away. For maintaining tasks with the inner and outer layers, users can extend the database, file system and related APIs along with data volume, user number and workflow complexity.

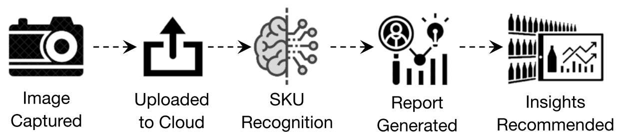
5 Case study

To verify the usability of the proposed framework, we introduce two real use cases whereby their ML lifecycle were initialized by MLife. We demonstrate its usage in ML systems for cloud-based product recognition (Clobotics 2021) and edge-based driver monitoring (Horizon Robotics 2021) in the domain of smart retail and transportation, respectively.

5.1 SKU recognition in cloud

An SKU (Stock Keeping Unit) refers to a distinct item for sale in a retail business, such as a product or service (Sawaya and Giauque 1986). SKU recognition is one of the core technologies that is applicable in smart retail scenarios. A typical workflow for smart retail is presented in Fig. 14. Briefly, it aims to capture images in retail stores using an application (APP) in a smart phone, upload these images to the cloud which recognizes the SKU class; the system then generates a variety of reports and finally deliver the reports and business insights to clients. However, ML models for SKU recognition are sensitive to product view and camera poses. Moreover, different brands and products of retail items may appear quite differently when they are tightly packed together in shelves and bays. Some "sub-brands" (commonly just a difference in packaging design, or flavour or version) are often distinguishable only by fine-grained packaging differences. Even for the same sub-brand, the design and packaging appearance can differ in different cities and seasons. Thus, ML models of a SKU recognition

Fig. 14 A typical workflow in smart retail scenarios



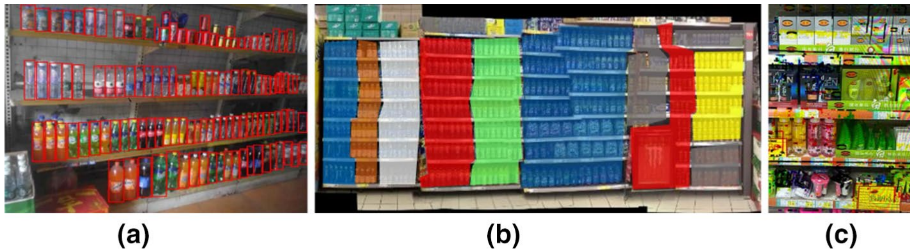


Fig. 15 Sample ML tasks/algorithms in a typical SKU recognition system: **a** Recognition of SKUs in an image. **b** Segmentation of shelves and bays. **c** Classification of unqualified images (e.g. moire patterns)

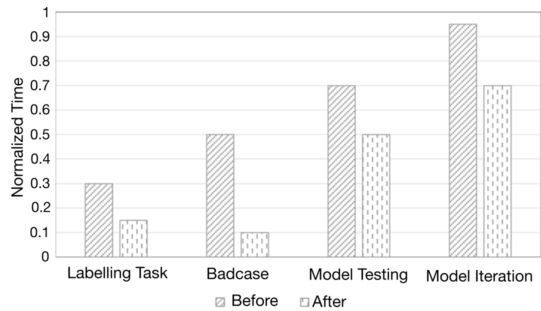


Fig. 16 Comparison of time cost for labelling task creation, badcase management, model testing and model iteration, before and after initialization using MLife

system should be iteratively updated to fit these ever-changing variations. Figure 15 presents some ML tasks that are typically involved in the entire SKU recognition system. Without the employment of MLife, most of these models are likely to be manually operated in each iteration, which is rather inefficient, tedious, and with no proper mechanism to cater for badcases that arose throughout the pipeline.

In Fig. 16, we compare some operations in the ML lifecycle before and after applying initialization using MLife. Particularly, we recorded the mean time taken by several known operations based on 5 model iterations and then normalized the results for comparison. We can clearly observe the improvement in efficiency. For instance, since training data is an operation which is possibly recorded by different teams, at different time and place, they are usually stored in different folders in the data store. In such a case, it is inefficient to manually find the specific required data and create labelling tasks. After the initialization using MLife, all data are managed in DAM in the CSV format and users can easily extend the APIs in the outer layer to connect with annotation systems so as to enable the automatic creation of labelling tasks. Specifically, this would allow the annotation column in CSV files to be filled after labelling. Based on our statistics, the time needed for the labelling task (including data gathering, uploading, cleaning and task creation) is reduced by around 50%. For badcase management, though existing tools such as gitlab (Engwall and Roe 2020) and JIRA (Ortu et al. 2015) can functionally meet the requirements, they are still not efficient enough and tailored for collaboration between teams and handling various operations shown in Fig. 6. This is because they have limited ability to (1) remove duplicated badcases from different testing teams and customers, (2) apply complicated statistics and analysis, and (3) provide APIs to adapt different domains, data formats, visualization and statistical requirements, etc. Theoretically, it is more efficient if a badcase management system is in conjunction with DAM since ML model iteration is driven by the existence of badcases in data flow. A main benefit of MLife is that all

badcases are managed together in BAM and they can be easily visualized, retrieved, updated and analysed so as to guide data collection and labelling in DAM as well as model testing in MTT. We report a total savings of around 80% in badcase management time. For model testing, the time cost is reduced by around 28% since the testing data can be prepared in MTT with only limited work done on data analysis via web pages. Moreover, the time cost can be further reduced by extending the APIs of MTT in the outer layer for automatic testing. Building gradually upon the improved efficiency at each stage, we can finally save around 26% on overall time cost for each model iteration with the initialization of the ML lifecycle using MLife.

5.2 Driver monitoring in edge

A Driver Monitoring System (DMS) aims to monitor the driving status of a driver so as to provide necessary assistance for safe and comfortable driving (Khan and Lee 2019). Typically, such monitoring systems are mainly focused on properties (e.g. gender, age, dressing, etc.), behaviours (smoking, calling, distracting, drinking, hands-off-the-wheel, etc.), expressions and drowsiness levels of a driver using vision-based approaches with infrared (IR) and RGB cameras. Recently, microphones are also employed in DMS since some audio-visual approaches (Srinivasan et al. 2019; Xu et al. 2019) are introduced to improve the experience of multimodal human-vehicle interaction. To ensure the real-time capability of the DMS, signals (both video and audio) are normally processed on the edge side. Specifically, as shown in Fig. 17, signals are captured and delivered to local processors for pre-processing and recognition. The recognition results are filtered and then combined with the results from the previous frames. Using pre-defined strategies and conditions, the result sequence is analysed in order to trigger different feedback to the driver. For instance, taking action such as turning on the air conditioner and playing music when visual signs of drowsiness (e.g. yawning and prolonged eye closure, etc.) in a driver are detected. Other actions include opening windows and turning down the volume when smoking and calling behaviours are detected, respectively.

To address the recognition tasks above, there are multiple ML models involved in DMS. As shown in Fig. 18, these models ranged from detection and regression to classification tasks. For example, the detection of facial landmarks is employed to determine the mouth and eye status for the purpose of classifying a driver's drowsiness level. However, there are various real-world challenges that are observed in practice after the deployment of DMS. Take vision-based smoking detection as an example (Fig. 19a, b)—light beams from windows and eating cylindrical shaped food or snacks are easily mistakenly recognized as that belonging to a smoking action. Meanwhile, drowsiness detection could potentially fail due to the influence of light spots on glasses and occlusion of eyes caused by the glass frame position (Fig. 19c). Since such scenarios are barely considered and covered in the initial iteration of ML models, it is meaningful to initialize the ML lifecycle so that these badcases can be timely observed and effectively handled in the subsequent iterations.

Motivated by the complications introduced in Sect. 2.2, Table 2 illustrates a comprehensive comparison between before and after initialization using MLife, on several ML

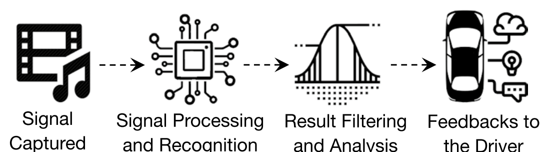


Fig. 17 A typical workflow in driver monitoring system scenarios

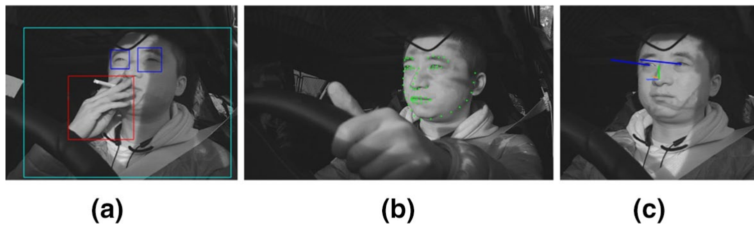


Fig. 18 ML tasks involved in DMS. **a** Body part/element (body, head, hand, eyes) detection. **b** Facial landmark extraction. **c** Gaze and head pose regression

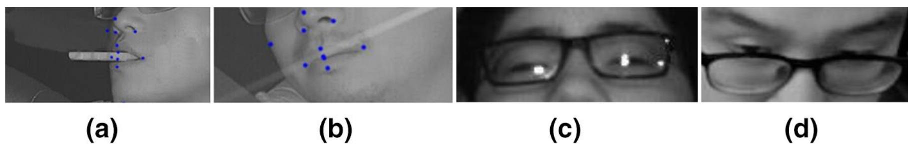


Fig. 19 Challenges of DMS: **a** Indistinguishable actions between eating and smoking. **b** Illusion from light beams. **c** Influences from light spots. **d** Occlusion from glass frame

lifecycle operations. We observe that operations among the ML lifecycle stages have drastically improved in terms of workflow efficiency and simplicity, standardization and scalability. Most manual works are replaced by semi-automatic operations accessible by users via web pages in the BAM, DAM, MTT and MSM modules. Moreover, MLife also ensures that the naming and storing of data name across different places are more standardized than traditional unconstrained strategies. Finally, MLife allows a far more convenient and systematic way of extending the ML lifecycle in terms of scenarios and workflow complexity among the operations.

To quantitatively evaluate the efficiency of the ML lifecycle and to demonstrate the improvement brought upon by MLife, we count the deployed model numbers of two customers before and after applying the initialization (see Fig. 20). Note that in this experiment based on the DMS, the major functions of two customers are similar but their accuracy requirements are different. As such, the number of deployed models are normalized for fair comparison. The right graph shows that the number of deployed models decreases over time with certain predictability, while the left shows that the number fluctuates significantly therefore unpredictable. In other words, the deployed model numbers are more stable using MLife, demonstrating the effectiveness of MLife in managing badcases, the use of data and training/testing models. Moreover, the workflow between data, algorithm and testing teams are gradually standardized in the process. With MLife, we also observe that the overall trend of the number of deployed models are also gradually reducing among the first five weeks before stabilizing at a certain level of need. Without MLife, the overall trend of deployed models is quite unpredictable; we observe that there were no models deployed in certain weeks—the second, third and sixth weeks before the initialization. Particularly, Fig. 20(right) presents the normalized number of expression-related badcases introduced in Sect. 3. We can clearly observe the rapid increase of badcases in the week 7. A strong benefit of using MLife is that the performance of expression recognition stabilizes much more within the model iteration process. The main reason is that MLife can effectively incorporate *human-in-the-loop* in terms of timely badcase alarming, root reason

Table 2 Comparison of operations among stages before and after the initialization of ML lifecycle using MLife

Before Initialization	After Initialization
Tools: Excel and JIRA issues	**Tools:** BAM in MLife
Workflow: At least 4 meetings between the Testing, Engineering and Algorithm teams to align badcases and discuss the plan for data collection and ML model iteration	**Workflow:** The Testing Team uploads all badcases in BAM. At most 2 meetings between the Engineering and Algorithm teams to discuss the plan for data collection and ML model iteration
Standardization: Hard to unify badcase names and priorities	**Standardization:** All badcase related metrics are standardized
Scalability: At most 3 customers due to redundant manual works on badcase management and alignment.	**Scalability:** Unlimited by extending the APIs in the outer layer
Tools: Excel and folders in services	**Tools:** DAM in MLife
Workflow: Manual moving, copying and analysis using Excel and Linux commands	**Workflow:** Upload the raw images to MLife and then carry out management and analysis in DAM using the built-in functions
Standardization: Hard to reuse the scripts for data analysis since algorithm engineers have different places to store data, and different strategies to name data	**Standardization:** All built-in functions and APIs can be reused and extended by algorithm engineers and data scientists from different teams
Scalability: At most 4 customers due to redundant manual works on data management and preparation.	**Scalability:** Unlimited by extending the APIs in the outer and inner layers
Tools: Python scripts and WIKI	**Tools:** DAM, MTT and MSM in MLife
Workflow: Copy all the data to a specific location manually or using scripts. After packaging, the training and testing tasks can be triggered. Training and testing results need to be managed manually in different folders and WIKI pages	**Workflow:** All training and testing data are stored in DAM and linked with the training and testing scripts via CSV IDs. After that, the training logs and testing results are stored in MTT. The trained ML models are stored in MSM and linked with MTT via model IDs
Standardization: Hard to standardize name and stored places of trained ML models, logs and testing results	**Standardization:** Name and stored places of ML models, logs, and testing results are all standardized
Scalability: At most 4 customers due to redundant manual works on data management and preparation.	**Scalability:** Unlimited by extending the APIs in the outer and inner layers
Tools: Excel and folders in services	**Tools:** MSM in MLife
Workflow: Manual moving and copying of ML models from different services for serving. Send emails to relevant people regarding the serving details	**Workflow:** All ML models and serving logs are stored and managed in MSM. Serving actions can be configured and triggered in MSM. Serving details are automatically sent to the relevant people afterwards
Standardization: Hard to standardize the name and stored place of ML models. Hard to automate the serving workflow	**Standardization:** Name and storing places are standardized. Serving details and workflow are unified after certain configurations
Scalability: At most 6 customers due to redundant manual works on ML model management and testing after serving.	**Scalability:** Unlimited and automated by extending APIs in the two layers

Top row: Badcase management and analysis; Second row: Data Management and Analysis; Third row: Model training and testing; Bottom row: Model management and serving. Note that the number of customers is estimated based on the affordability of a fixed number of Algorithm, Engineering and Testing team members

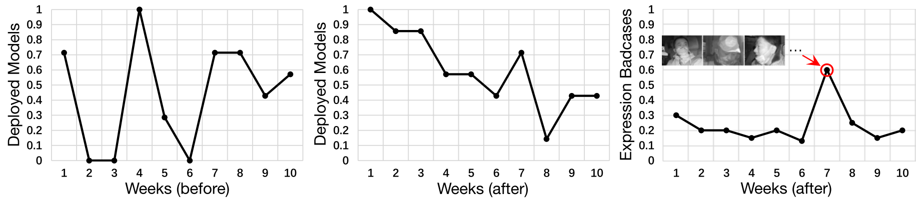


Fig. 20 Comparison of the normalized number of deployed models per week before (left) and after (middle) the initialization of ML lifecycle using MLife. Normalized number of expression-related badcases is detailed in the right figure. The red mark indicates the start of pandemic

and optimal solution exploration, semi-automatic training, testing and serving, etc. Thus, we are able to conclude that MLife can effectively improve the robustness of ML systems particularly in how the ML lifecycle is managed and handled.

6 Conclusion

In this paper, we introduce a simple yet efficient framework, MLife, for quick and effective initialization of the ML lifecycle. MLife is motivated by the need for operations in the ML lifecycle to be re-organized into inner, middle and outer layers where the middle layer contains crucial, standardized operations for initialization while the inner and outer layers provide APIs and tools that are extensible to various scenarios, data volume and user number after initialization. Therefore, MLife can be used for the purpose of both initialization and maintenance. Intuitively, MLife thrives on the fact that ML systems are largely driven by badcases and hence, the proposal of BAM, DAM, MTT and MSM stages for iteratively circulating the workflow from badcases back to the data and ML models. We present two real-world use cases to evaluate the usability of MLife on cloud- and edge-based ML systems. Our comparative (before-after) results show that MLife can effectively improve the efficiency of ML model iteration and workflow standardization.

Acknowledgements The work is supported by the funding from Clobotics and Horizon Robotics under the Research Program of Smart Retail and Driver Monitoring System, respectively, and in part by CREST R&D Grant T03C1-17, Malaysia.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

5Analytics. Retrieved from 08 May 2021. https://www.5analytics.com/
airflow. Retrieved from 08 May 2021. https://airflow.apache.org/
Algorithmia. Retrieved from 08 May 2021. https://algorithmia.com/
Amazon, (2020). Training ml models. In *Amazon machine learning: Developer guide* (pp. 72–73). Amazon Web Services.
Amazon web services. Retrieved from 08 May 2021. https://aws.amazon.com/
Ashmore, R., Calinescu, R., & Paterson, C. (2019). *Assuring the machine learning lifecycle: Desiderata, methods, and challenges*. arXiv preprint arXiv:1905.04223

Aslam, F. A., Mohammed, H. N., Mohd, J. M., Gulamgaus, M. A., & Lok, P. (2015). Efficient way of web development using python and flask. *International Journal of Advanced Research in Computer Science, 6*(2), 54.

Baylor, D., Breck, E., Cheng, H. T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., & Koo, C. Y. (2017). Tfx: A tensorflow-based production-scale machine learning platform. In *ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 1387–1395).

Bengio, S. (2015). Sharing representations for long tail computer vision problems. In *ACM on international conference on multimodal interaction* (p. 1).

Bhosale, S., Patil, T., & Patil, P. (2015). Sqlite: Light database system. *International Journal of Computer Science and Mobile Computing, 4*(4), 882.

Chen, C., Golshan, B., Halevy, A., Tan, W., & Doan, A. (2018). Biggorilla: An open-source ecosystem for data preparation and integration. *IEEE Data Engineering Bulletin, 41*(2), 10–22.

Clobotics: Cloud image recognition. Retrieved from 08 May 2021. https://clobotics.com/retail

Cortex. Retrieved from 08 May 2021. https://www.cortex.dev/

craft ai. Retrieved from 08 May 2021. https://www.craft.ai/

Crankshaw, D., Wang, X., Zhou, G., Franklin, M., Gonzalez, J., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. In *USENIX symposium on operating systems design and implementation (OSDI)* (pp. 613–627).

Datatron. Retrieved from 08 May 2021. https://www.datatron.com/

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *IEEE conference on computer vision and pattern recognition* (pp. 248–255).

Engwall, K., & Roe, M. (2020). Git and GitLab in library website change management workflows. *Code4Lib Journal, 48*. https://journal.code4lib.org/articles/15250.

Fan, J., & Li, G. (2018). Human-in-the-loop rule learning for data integration. *IEEE Data Engineering Bulletin, 41*(2), 104–115.

Fanelli, D., & Piazza, F. (2020). Analysis and forecast of covid-19 spreading in China, Italy and France. *Chaos, Solitons & Fractals, 134*, 109761.

FBLearner. Retrieved from 08 May 2021. https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/

Flyte. Retrieved from 08 May 2021. https://lyft.github.io/flyte/

Horizon Robotics: Driver monitoring system. Retrieved from 08 May 2021. https://en.horizon.ai/product/nebula

JupyterHub. Retrieved from 08 May 2021. https://jupyter.org/hub

Khan, M. Q., & Lee, S. (2019). A comprehensive survey of driving monitoring and assistance systems. *Sensors, 19*(11), 2574.

KNIME. Retrieved from 08 May 2021. https://www.knime.com/

kubeflow. Retrieved from 08 May 2021. https://www.kubeflow.org/

Lee, D., Macke, S., Xin, D., Lee, A., Huang, S., & Parameswaran, A. (2019). A human-in-the-loop perspective on automl: Milestones and the road ahead. *IEEE Data Engineering Bulletin, 42*(2), 59–70.

Lee, Y., Scolari, A., Chun, B., Santambrogio, M., Weimer, M., & Interlandi, M. (2018). Pretzel: Opening the black box of machine learning prediction serving systems. In *USENIX symposium on operating systems design and implementation (OSDI)* (pp. 611–626).

Lee, Y., Scolari, A., Chun, B., Weimer, M., & Interlandi, M. (2018). From the edge to the cloud: Model serving in ml.net. *IEEE Data Engineering Bulletin, 41*(4), 46–53.

Li, S., & Deng, W. (2020). Deep facial expression recognition: A survey. *IEEE Transactions on Affective Computing*. https://doi.org/10.1109/TAFFC.2020.2981446

Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing, 234*, 11–26.

Miao, H., Li, A., Davis, L., & Deshpande, A. (2017). Modelhub: Deep learning lifecycle management. In *International conference on data engineering* (pp. 1393–1394).

Michelangelo. Retrieved from 08 May 2021. https://eng.uber.com/michelangelo/

Microsoft. Retrieved from 08 May 2021. https://docs.microsoft.com/en-us/azure/machine-learning/

Microsoft machine learning server. Retrieved from 08 May 2021. https://docs.microsoft.com/en-us/machine-learning-server

mlflow. Retrieved from 08 May 2021. https://mlflow.org/docs/

mxnet. Retrieved from 08 May 2021. https://mxnet.cdn.apache.org/

Mxnet model server (mms). Retrieved from 08 May 2021. https://github.com/awslabs/mxnet-model-server

NiFi. Retrieved from 08 May 2021. https://nifi.apache.org/

Olston, C., Li, F., Harmsen, J., Soyke, J., Gorovoy, K., Lao, L., Fiedel, N., Ramesh, S., & Rajashekhar, V. (2017). Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML systems at NIPS 2017* (pp. 1–8).

Ortu, M., Destefanis, G., Kassab, M., Counsell, S., Marchesi, M., & Tonelli, R. (2015). Would you mind fixing this issue? In *International conference on Agile software development* (pp. 129–140). Springer.

Pan, J., & McElhannon, J. (2018). Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal, 5*(1), 439–449.

Peltarion. Retrieved from 08 May 2021. https://peltarion.com/

Polyzotis, N., Roy, S., Whang, S., & Zinkevich, M. (2018). Data lifecycle challenges in production machine learning: A survey. *ACM SIGMOD Record, 47*(2), 17–28.

Pytorch. Retrieved from 08 May 2021. https://pytorch.org/

Raschka, S., & Mirjalili, V. (2019). *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing Ltd.

Russell, B., Torralba, A., Murphy, K., & Freeman, W. (2008). Labelme: A database and web-based tool for image annotation. *International Journal of Computer Vision, 77*(1–3), 157–173.

SageMaker. Retrieved from 08 May 2021. https://aws.amazon.com/cn/sagemaker/

SAS: Sas model manager. Retrieved from 08 May 2021. https://www.sas.com/en_us/software/model-manager.html

Sawaya, W., & Giauque, W. (1986). *Production and operations management*. Harcourt Brace Jovanovich.

Schelter, S., Bießmann, F., Januschowski, T., Salinas, D., Seufert, S., & Szarvas, G. (2018). On challenges in machine learning model management. *IEEE Data Engineering Bulletin, 41*(4), 5–15.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. In *International conference on neural information processing systems* (pp. 2503–2511).

Seldon. Retrieved from 08 May 2021. https://www.seldon.io/

Srinivasan, T., Sanabria, R., & Metze, F. (2019). *Analyzing utility of visual context in multimodal speech recognition under noisy conditions*. arXiv preprint arXiv:1907.00477

Tensorflow serving. Retrieved from 08 May 2021. https://www.tensorflow.org/serving

valohai. Retrieved from 08 May 2021. https://valohai.com/

Vartak, M., & Madden, S. (2018). Modeldb: Opportunities and challenges in managing machine learning models. *IEEE Data Engineering Bulletin, 41*(4), 16–25.

Xu, H., Zhang, H., Han, K., Wang, Y., Peng, Y., & Li, X. (2019). *Learning alignment for multimodal emotion recognition from speech*. arXiv preprint arXiv:1909.05645

Zaharia, M., et al. (2018). Accelerating the machine learning lifecycle with mlflow. *IEEE Data Engineering Bulletin, 41*(4), 39–45.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.