Demonstration of the Ease.ml System for MLDev and MLOps

Bojan Karlaš[*,†], Wentao Wu[§], Ce Zhang[†]
[†]ETH Zurich, [§]Microsoft Research
[†]{bojan.karlas, ce.zhang}@inf.ethz.ch, [§]wentao.wu@microsoft.com

ABSTRACT

We demonstrate `ease.ml`, an end-to-end lifecycle management system for MLDev and MLOps. Unlike other work on AutoML that tries to improve only the ML pipeline (e.g., model training, testing, and hyperparameter tuning), `ease.ml` aims for *jointly* optimizing the ML pipeline *as well as* the related data management and software engineering procedures that take place beforehand and afterwards (e.g., feasibility study, data cleaning, and continuous integration). Specifically, `ease.ml` introduces a novel management process that defines necessary steps that most ML application developers would have to undergo. It also provides effective management tools based on novel techniques that address specific challenges raised in each individual step. In this paper, we present the functionalities that `ease.ml` features, highlight its key design principles and implementation details, as well as showcase its efficacy via a variety of real-world use cases and scenarios.

PVLDB Reference Format:
Bojan Karlaš[*,†], Wentao Wu[§], Ce Zhang[†] . Demonstration of the Ease.ml System for MLDev and MLOps. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The past decade has witnessed the explosion of machine learning (ML) applications. The ongoing movement towards the so-called "software 2.0" further offers a glimpse of the future where most software programs could include ML components learned from data [9]. While this technological drift is taking place, developers are, on the other hand, facing new challenges in terms of building ML applications with satisfactory quality. Unlike traditional software engineering, which is governed by a set of well-known principles that have been practiced for decades, the development of ML applications currently lacks such principles and the quality largely depends on the developer's own expertise.

In the past several years, we have been working on this new direction of bringing software engineering principles for ML application development. We built `ease.ml`, to the best of our knowledge, the first end-to-end lifecycle management system for application developers that are not presumed ML experts. Unlike other work on AutoML that focuses on only the ML pipeline of an application (e.g., training and testing ML model, hyperparameter tuning), `ease.ml` takes care of the *end-to-end pipeline* that also includes necessary prerequisite procedures (such as feasibility analysis and data cleaning) and quality insurance procedures afterwards (such as continuous integration), in an automated manner. Specifically, as illustrated in Figure 1, `ease.ml` introduces a novel management process that covers necessary steps that most developers would have to undergo throughout the lifecycle of an ML application, and each individual step is supported by corresponding `ease.ml` components/tools based on novel techniques developed in our previous and ongoing works [2–8, 10, 15, 17–20, 23]. An in-depth overview of the `ease.ml` process has also been given in [11].

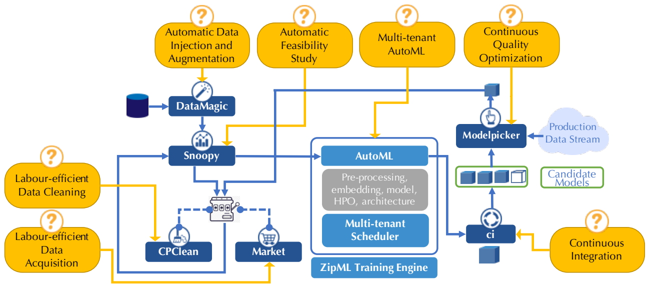

Automatic Data Injection and Augmentation
Automatic Feasibility Study
Multi-tenant AutoML
Continuous Quality Optimization
DataMagic
Snoopy
Modelpicker
Production Data Stream
Labour-efficient Data Cleaning
AutoML
Pre-processing, embedding, model, HPO, architecture
Multi-tenant Scheduler
Candidate Models
Labour-efficient Data Acquisition
CPClean
Market
ci
Continuous Integration
ZipML Training Engine

Figure 1: Ease.ML Process

In this paper, we primarily focus on demonstrating the functionalities of the `ease.ml` system itself, using a variety of real-world use cases and scenarios. We also briefly describe some key design decisions and implementation details of our system.

1.1 An Illustrative Example

Alice, a data scientist, was given a task of developing an ML model that can predict whether a customer will return based on the feedback collected. She was given a tabular dataset with various columns obtained from feedback forms. She immediately loaded the data, split it into training and validation subsets, and passed them into `auto-sklearn` [1], an off-the-shelf AutoML tool, with the goal of finding some model with desired accuracy. After many hours of running `auto-sklearn`, she encountered her first challenge.

Challenge 1: The test accuracy was always below 70%, even though `auto-sklearn` had tried 15 different models with many different hyperparameter configurations. What was causing this?

Solution: Alice believed that the dataset has problems, which may explain the low accuracy observed, but she was not sure what actions should be taken. She had to spend several weeks manually cleaning the entire dataset. She then applied AutoML again and this time obtained a model with 80% accuracy.

Pitfall: Alice had to make decisions by herself on whether to clean the data or not (where her decision is *yes*) and which data examples to clean (where her decision is to clean *all of them*). Such decisions may be sub-optimal and she may waste a lot of time. □

Alice deployed the model in production. Six months later, her team collected more training data, so she had to revisit her task and encountered a new challenge.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Challenge 2: Several new models had emerged in the meantime that were applicable to her problem. Could the prediction system be improved with the newly available models?

Solution: Alice ended up updating her AutoML system by training the new models on the updated dataset. She even had to implement one model by herself. The AutoML process took much longer due to more models being examined. Afterwards, she observed that the new test accuracy is 82%. Since this meant 2% improvement, she decided to deploy the new model in production.

Pitfall: Overcoming this challenge again involved many decisions and manual efforts, all of which could have been done automatically if an appropriate MLOps system was put in place. The availability of new data and new models could have automatically triggered a rerun of AutoML. Furthermore, the decision to favor the new model may be incorrect, since the 2% accuracy difference may be just within the margins of accuracy estimation errors. □

After another six months, the QA team discovered that in summer months the accuracy of Alice's model was indeed above 80% as expected, but in winter months it could drop below 70%. After a lot of web searches and asking colleagues for help, Alice learned that this is a typical data drifting scenario. She then discovered that the model she previously trained actually outperforms the new model in winter months. This posed yet another challenge for her.

Challenge 3: How to ensure the quality of production models over a long period of time when data drift can happen?

Solution: Alice ended up setting a simple rule to deploy the old model in winter months and the new model in summer months.

Pitfall: Although this solution might be fine in the short term, it may not be optimal, especially for months that are in-between. It is really hard to make the right decision without evaluating both models with fresh production data. □

Typical ML workflows that data scientists are dealing with in their daily life pose many *common pitfalls*, as illustrated in the above example. Yet, a comprehensive solution is still missing even today. `ease.ml` is our current best attempt at defining an abstraction for automatically managing such end-to-end ML workflows to help data scientists avoid the aforementioned common pitfalls. In the rest of this paper, we outline the main functional components of `ease.ml` and present demonstration scenarios that illustrate how it could serve Alice in overcoming the pitfalls that she encountered.

2 THE EASE.ML SYSTEM

As described in [11], `ease.ml` adopts a unified logical view of ML datasets that can be summarized as a *probabilistic database* with *typed* records. It offers a unified namespace where instances of datasets can be stored and recovered from. It allows to perform various data transformations on datasets to produce new datasets.

On top of this foundation, `ease.ml` comes with an array of built-in tools that we have developed over the years and have presented in prior work [2–8, 10, 15, 17–20, 23]. These tools focus on specific aspects when dealing with ML workflows. The ML workflow itself is modeled as a data processing graph that the user constructs by interacting with `ease.ml`, where the graph is also treated as data that can be stored in the unified namespace and retrieved on demand. Finally, `ease.ml` provides a facility called `whatnext`, which can offer suggestions to the user on which further steps could be beneficial given the current state of the ML workflow.

2.1 Components and Tools in Ease.ml

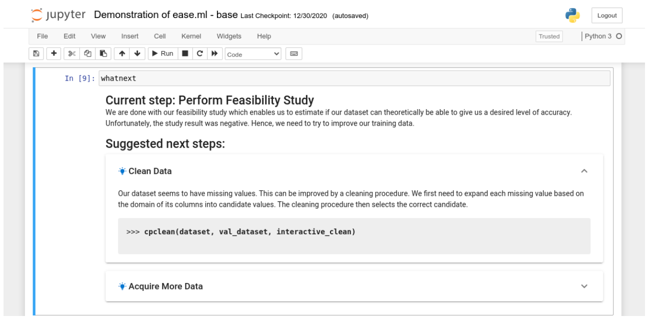
We provide a listing of the tools offered in `ease.ml` that support the individual steps defined in the `ease.ml` process (ref. Figure 1 and [11]), along with a summary of their inputs and outputs:

① `datamagic`	
Purpose:	Data augmentation.
Inputs:	Input dataset.
Outputs:	Augmented dataset.
② `snoopy`	
Purpose:	Feasibility study to check if the expected accuracy is achievable with a given dataset.
Inputs:	Augmented dataset; Expected accuracy.
Outputs:	A `yes/no` outcome of the feasibility study.
③ `cpclean`	
Purpose:	Iteratively perform cleaning of data points prioritized by the impact on the validation accuracy.
Inputs:	Augmented dataset.
Outputs:	Sufficiently cleaned dataset.
④ `market`	
Purpose:	Iteratively acquire data labels based on the task-specific value of data points.
Inputs:	Augmented dataset.
Outputs:	Newly acquired labelled data points.
⑤ `automl`	
Purpose:	Perform model and feature search based on validation accuracy.
Inputs:	Augmented, partially cleaned, dataset.
Outputs:	Set of candidate models sorted by the validation accuracy they achieved.
⑥ `mltest`	
Purpose:	Check if the model passes the test condition while avoiding overfitting the test set.
Inputs:	Candidate model; Production model pool; Test data pool; Test condition.
Outputs:	Candidate model if it passes the test condition, otherwise empty set.
⑦ `modelpicker`	
Purpose:	Pick model with the best performance on fresh production data.
Inputs:	Production model pool; Fresh production data.
Outputs:	Best performing model.

2.2 Recommendation on Next Ease.ml Steps

It can be quite hard to manage the complexity of ML workflows, especially for less experienced data scientists. There are lots of decisions to be made and each sub-optimal decision leads to time being spent on exploring paths that will not be fruitful.

To ease the burden on the user, `ease.ml` further provides a command called `whatnext`. It analyzes the current state of the ML workflow and offers recommendations about sensible next steps that the user can take. Currently, we implement `whatnext` as a simple rule-based recommendation engine, but it can be augmented in the future. Figure 2 presents the interactive UI that pops up when the user invokes the `whatnext` command.


jupyter Demonstration of ease.ml - base Last Checkpoint: 12/30/2020 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [9]: whatnext
```

Current step: Perform Feasibility Study

We are done with our feasibility study which enables us to estimate if our dataset can theoretically be able to give us a desired level of accuracy. Unfortunately, the study result was negative. Hence, we need to try to improve our training data.

Suggested next steps:

Clean Data

Our dataset seems to have missing values. This can be improved by a cleaning procedure. We first need to expand each missing value based on the domain of its columns into candidate values. The cleaning procedure then selects the correct candidate.

```
>>> cpclean(dataset, val_dataset, interactive_clean)
```

Acquire More Data

Figure 2: The interface of the `whatnext` command.

3 DESIGN AND IMPLEMENTATION

In this section, we highlight a few design choices and implementation details when building `ease.ml`.

3.1 Data Model and Query

The core data model of `ease.ml` is *relational*. Each object is a set X that is an instance of some relation $R := \{c_1 : T_1, c_2 : T_2, \ldots\}$, where $c_1, c_2, \ldots$ are column names and $T_1, T_2, \ldots$ are column types. A type T can be a simple primitive (e.g., string or integer) but can also be more complex, such as a nested relation. Thus, objects are sets of named and typed tuples. For example, we can have a relation $\texttt{Persons} := \{\texttt{name} : \texttt{String}, \texttt{age} : \texttt{Int}, \texttt{houses} : \texttt{House}\}$, where `House` is another relation. We list several other features of our data model in the remainder of this section.

Modeling uncertainty. We observed that ML datasets often involve uncertainty, for example when performing data augmentation or cleaning. To model this, we adopt probabilistic relational semantics in our data model. Hence, each tuple $t \in X$ is associated with a set of *candidate* tuples $t := \{t_1, t_2, \ldots\}$, referred to as a *block*. The semantics of each tuple is that it evaluates to a single candidate chosen at random, based on some distribution. For example, we may not be sure about the age of a person, so we can store it as a set of candidate values, each associated with some probability.

Computational operators. We use the same relational model to represent our computational operators. A relation can be "invoked" to produce a *query*. For example, $\texttt{Persons}(\texttt{age} = 30)$ would define a query of all people with age 30. Running that query would require a database lookup. Furthermore, we can have a relation associated with an arbitrary function. For example, any of the `ease.ml` components (e.g., `automl`) is a relational operator. Hence, invoking

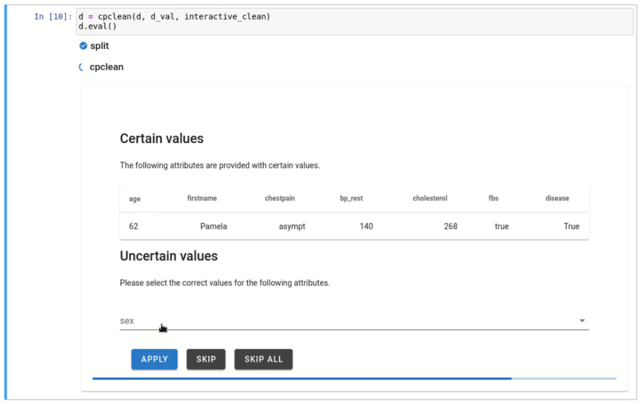
$$\texttt{automl}(\texttt{d\_tr} = D_{tr}, \texttt{d\_val} = D_{val}, \texttt{models} = M)$$

defines a query that tries to find a model $m \in M$ which returns the best score when trained on D_{tr} and evaluated on D_{val}.

Subtyping. Given that *relations are types*, a query Q over a relation R can be viewed as a subtype $Q \subseteq R$. All types (relations and queries) can thus be stored in a *unified* hierarchical namespace. Executing a query is thus equivalent to computing an instance of the type Q. This subtyping system is useful to ensure *type safety* of the composition of computational operators.

3.2 Workflow Construction and Execution

An ML workflow includes a lot of exploration. Most explored paths lead to dead ends and can be thrown away. Few may turn out to be



```
In [10]: d = cpclean(d, d_val, interactive_clean)
         d.eval()
```

Certain values

The following attributes are provided with certain values.

age	firstname	chestpain	bp_rest	cholesterol	fbs	disease
62	Pamela	asympt	140	268	true	True

Uncertain values

Please select the correct values for the following attributes.

sex

Figure 3: The interactive labeling interface.

promising, and we want to store them so that we can revisit them later and reproduce their results. However, it is almost impossible to know the promising paths *beforehand*. An ideal user interface would give us maximum flexibility and minimum overhead in the beginning, while allowing us to store and reuse already explored paths when running the ML workflow.

We design our workflow mechanism with the above goals in mind. We treat the *query* as a central object for performing any form of computation in `ease.ml`. A query can be iteratively constructed so it implicitly carries information about its entire *lineage*. We provide various facilities for managing queries.

Composition. Queries are *composable* objects that describe a *result set*. As such, they can be included in other queries to form an augmented workflow graph.

Analysis. We can inspect the structure of the execution graph in order to determine the state of the workflow. We use this analysis to implement the aforementioned `whatnext` command that suggests potential next steps that might be fruitful.

Storage. Queries can be stored inside a unified hierarchical namespace just like any other object.

Execution. Finally, the result set of a query can be materialized. This result set can be cached for future reuse. If a query references other queries, we first check if the corresponding results are cached before executing the referenced queries.

3.3 User Interaction

To ensure familiarity for data scientists and interoperability with most ML tools, we choose Python as the main programming language for users to interact with `ease.ml`. We provide a rich set of functions that can be used for accessing object storage, data inspection, query construction and execution.

For maximum interactivity, we opt for Jupyter notebook as the programming environment, which makes it easy to experiment with different approaches and immediately visualize results.

One step towards this goal involves pluggable UI components that can be an integral part of the user's workflow. For example, a user may want to clean some data using the `cpclean` component provided by `ease.ml`. This component selects a minimal subset of dirty data points that can be cleaned interactively by the user. In Figure 3, we show an example of this interactive cleaning UI that pops below a regular code cell in the Jupyter notebook upon execution of the workflow graph.

4 DEMONSTRATION SCENARIOS

We demonstrate how `ease.ml` helps users like Alice to manage her journey along a typical data science workflow that, apart from model search, features data preparation and model management.

Step ① (Load data): Our workflow starts with loading the data. We recognize several types of datasets: (1) ***training and validation data*** (typically labeled, used for model training); (2) ***test data*** (most often labeled, used for testing the model against production models); and (3) ***production data*** (typically unlabeled, the freshest data used for picking the best production model).

Step ② (Invoke `whatnext`): We demonstrate how this helps us navigate the workflow. In Jupyter notebook it displays an interactive dialog that informs the user about suggested next steps.

Step ③ (Inject uncertainty): The first step is to pass our training data through `datamagic`, which can inject uncertainty into the data by expanding dirty values or by performing data augmentation. The output of this step is a probabilistic relation.

Step ④ (Feasibility study): Before we train models with our data, we need to check if our data can support our desired model accuracy. We use the `snoopy` component to estimate this.

Step ⑤ (Clean data): If the feasibility study returns `no`, we need to improve our data. The system offers us to do this either through cleaning or data acquisition. In this step, we will show how `cpclean` helps us clean our data by finding the *minimal* subset of data examples that we can interactively clean in order to be able to make consistent predictions (CP). After this step, we can go back to **Step ④** to check if we have sufficiently improved our dataset.

Step ⑥ (Run model optimization and obtain best model): If the feasibility study returns `yes`, then we can invoke `automl` to perform automated feature engineering, model selection and training, and hyperparameter tuning. The outcome of this step is a stream of models with their validation scores. We can then select the model with the best validation score, which now becomes our *candidate model* for production deployment. However, before that, we need to perform several integration steps to ensure that we deploy only the best performing models.

Step ⑦ (Load production pool of models): Since subsequent steps are about *integration* of the candidate model into the production pool, we need to load that model pool from storage. We can observe that a model is just like any other object, which can be stored inside a relation.

Step ⑧ (Perform continuous integration): The first step is testing the candidate model using the `mltest` component. A test is performed by using the test dataset to evaluate a given test condition. The `mltest` component automatically prevents us from *overfitting* the test dataset. This is done by swapping out the currently used chunk after we have reached the limit on the number of times that we can use the test set. If a candidate model passes the test, we can observe it getting added to the production model pool.

Step ⑨ (Pick best production model): The final task is to pick the model from the production pool that will most likely have the best performance on production data. Since this data is freshly acquired from production, it is typically *unlabeled.* We use the `modelpicker` component to help us interactively label the data examples that will be the most informative for determining which model is the best.

5 RELATED AND PRIOR WORK

There has been a lot of work on building abstractions for ML workflows. Examples include data acquisition with *weak supervision* [14], *debugging and validation* [12, 22], *model management* [13] and *deployment* [21], *knowledge integration* [24], and *data cleaning* [16]. The result is a powerful toolkit for various stages of ML workflow. However, the task of navigating the space of all the available tools and the plethora of choices that come with them remains an overwhelming challenge for today's ML application developers.

This work represents our latest attempt at a system for improving the usability of ML. Our previous attempt focused mainly on a specific part of an ML workflow: model selection and tuning, which we tried to solve in the multi-tenant setting [8, 10, 23]. In this work we have significantly broadened our scope to the entire ML workflow. Specifically we provide tools to support decision making on the user side and overall management of datasets and models. To achieve this, we take advantage of components that were the result of our previous work [2–8, 10, 15, 17–20, 23]. We presented the conceptual aspects of this new abstraction that we envision in [11], while here we focus on the implementation aspects as well as on a live demonstration of the capabilities of our system.

REFERENCES

[1] Feurer et al. 2015. Efficient and Robust Automated Machine Learning. In *NIPS.*

[2] Ruoxi Jia et al. 2019. Efficient Task-Specific Data Valuation for Nearest Neighbor Algorithms. *PVLDB* (2019).

[3] Ruoxi Jia et al. 2019. An Empirical and Comparative Analysis of Data Valuation with Scalable Algorithms. *arXiv* (2019).

[4] Ruoxi Jia et al. 2019. Towards Efficient Data Valuation Based on the Shapley Value *(AISTATS).*

[5] Mohammad Reza Karimi et al. 2020. Online Active Model Selection for Pre-trained Classifiers.

[6] Bojan Karlas et al. 2020. Building Continuous Integration Services for Machine Learning. In *KDD.*

[7] Bojan Karlaš et al. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *arXiv* (2020).

[8] Bojan Karlaš et al. 2018. Ease.Ml in Action: Towards Multi-Tenant Declarative Learning Services. *VLDB Demo* (2018).

[9] Andrej Karpathy. [n.d.]. Software 2.0. https://karpathy.medium.com/software-2-0-a64152b37c35.

[10] Tian Li et al. 2018. Ease.ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads. In *PVLDB.*

[11] Leonel Aguilar Melgar et al. 2021. Ease.ML: A Lifecycle Management System for MLDev and MLOps. In *Conference on Innovative Data Systems Research.*

[12] Akshay Naresh Modi et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD 2017.*

[13] Supun Nakandala et al. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *PVLDB* (2020).

[14] Alexander Ratner et al. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *PVLDB* (2017).

[15] Johannes Rausch et al. 2021. DocParser: Hierarchical Structure Parsing of Document Renderings. *AAAI* (2021).

[16] Theodoros Rekatsinas et al. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* (2017).

[17] Cedric Renggli et al. 2019. Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *SysML.*

[18] Cedric Renggli et al. 2019. Ease.ml/ci and Ease.ml/meter in Action: Towards Data Management for Statistical Generalization. In *VLDB Demo.*

[19] Cedric Renggli et al. 2020. Ease.ml/snoopy in Action: Towards Automatic Feasibility Analysis for Machine Learning Application Development. In *VLDB Demo.*

[20] Giuseppe Russo et al. 2020. Control, Generate, Augment: A Scalable Framework for Multi-Attribute Text Generation. *arXiv* (2020).

[21] Manasi Vartak et al. 2016. ModelDB: A System for Machine Learning Model Management. In *HILDA.*

[22] Weiyuan Wu et al. 2020. Complaint-Driven Training Data Debugging for Query 2.0. In *SIGMOD.*

[23] Chen Yu et al. 2019. AutoML from Service Provider's Perspective: Multi-device, Multi-tenant Model Selection with GP-EI *(AISTATS).*

[24] Ce Zhang et al. 2017. DeepDive: Declarative Knowledge Base Construction. *Commun. ACM* (2017).