Building Continuous Integration Services for Machine Learning

Bojan Karlaš[1,2], Matteo Interlandi[1], Cedric Renggli[2], Wentao Wu[1], Ce Zhang[2], Deepak Mukunthu Iyappan Babu[1], Jordan Edwards[1], Chris Lauren[1], Andy Xu[1], Markus Weimer[1]
[1]Microsoft, [2]ETH Zurich

ABSTRACT

Continuous integration (CI) has been a *de facto* standard for building industrial-strength software. Yet, there is little attention towards applying CI to the development of machine learning (ML) applications until the very recent effort on the theoretical side. In this paper, we take a step forward to bring the theory into practice.

We develop the first CI system for ML, to the best of our knowledge, that integrates seamlessly with existing ML development tools. We present its design and implementation details.

ACM Reference Format:
Bojan Karlaš[1,2], Matteo Interlandi[1], Cedric Renggli[2], Wentao Wu[1], Ce Zhang[2], Deepak Mukunthu Iyappan Babu[1], Jordan Edwards[1], Chris Lauren[1], Andy Xu[1], Markus Weimer[1] . 2020. Building Continuous Integration Services for Machine Learning. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394486.3403290

1 INTRODUCTION

Recent decades have witnessed an increasingly frequent usage of machine learning for a wide range of mission critical applications. However, one challenge in building such applications, especially from the perspective of practitioners and domain scientists, is *overfitting*. In our experience in supporting a range of industrial and academic users [9, 14, 22–25], it is not uncommon for users of modern ML platforms to suffer from this problem, as illustrated in Figure 1(a), that the gap between the estimated test/validation accuracy and true test accuracy increases during the development process. *Can we provide tools to help practitioners tackle this problem?*

In this paper, we draw our inspiration from continuous integration (CI), which has been part of the industry standard of modern software development [12], evidenced by the recent surge of cloud-hosted software development services such as Azure DevOps [3] and AWS CodePipeline [2]. CI services lift the burden of managing the software development lifecycle from the developers by providing a variety of tools for building, testing, and deploying software applications in an automated and iterative manner. Development of machine learning (ML) applications is not much different in this regard from regular software systems – it typically requires many iterations as developers try to continuously improve the quality of their ML models. *Can we build continuous integration services for ML to give users constant feedbacks and signals about overfitting?*

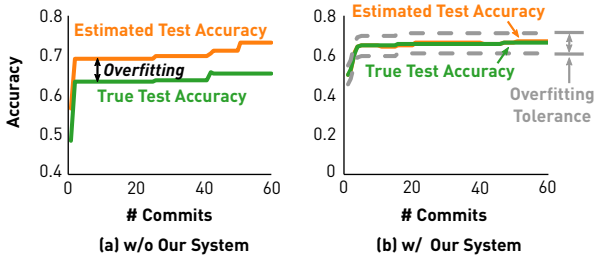

Figure 1: (a) The challenge of building a CI system for ML is that, if not being careful, one might overfit the test set when committing multiple models during the CI process; (b) The goal of our system is to provide rigorous guarantees on the overfitting behavior by, intuitively, measuring the "information leakage" from the test set during the CI process.

Challenges. None of the existing CI services are sufficient when it comes to ML applications [20]. One major issue lies in testing: In traditional software testing, test cases can be reused infinitely to evaluate test assertions, and simply return *deterministic* binary `true`/`false` signals. *In ML testing, every pass/fail signal leaks information about the test set itself and thus may lead to overfitting, resulting in false positive/false negative outcomes* (see Figure 1).

(Example) Consider the test assertion `n - o > 0.01`, where `n` and `o` represent the accuracy of the new and old version of an ML model in consecutive development iterations. The semantics of the test assertion is clear — the test will pass only if the new model improves accuracy by at least 0.01. To evaluate this test assertion, however, one needs to estimate the accuracy of both models using a test set, which is a finite set of i.i.d. samples from the underlying data distribution. Such estimates are inherently uncertain and so is the `true`/`false` evaluation result of the test assertion. Therefore, one has to interpret the evaluation result from a *probabilistic* view: The result holds with high probability if the test set is sufficiently large. Ensuring such probabilistic guarantees for ML test assertions, when the same test set is repeatedly used for evaluation, is one major challenge that "CI for ML" services need to address.

In this paper, we develop the first "CI for ML" service. As illustrated in Figure 1(b), our system controls the size of test/validation set such that the gap between the estimated accuracy and the true test accuracy is guaranteed to be smaller than a small constant specified by the user. To the best of our knowledge, this is the first such system that integrates seamlessly with existing ML development tools. Our service provides a framework for testing ML models that is based on strict theoretical bounds and enables a principled way to avoid over-fitting the test set, which is a common problem that is easily overlooked. Our service is also seamlessly integrable with existing CI frameworks such as TravisCI, Microsoft Azure DevOps,

KDD '20, August 23–27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403290

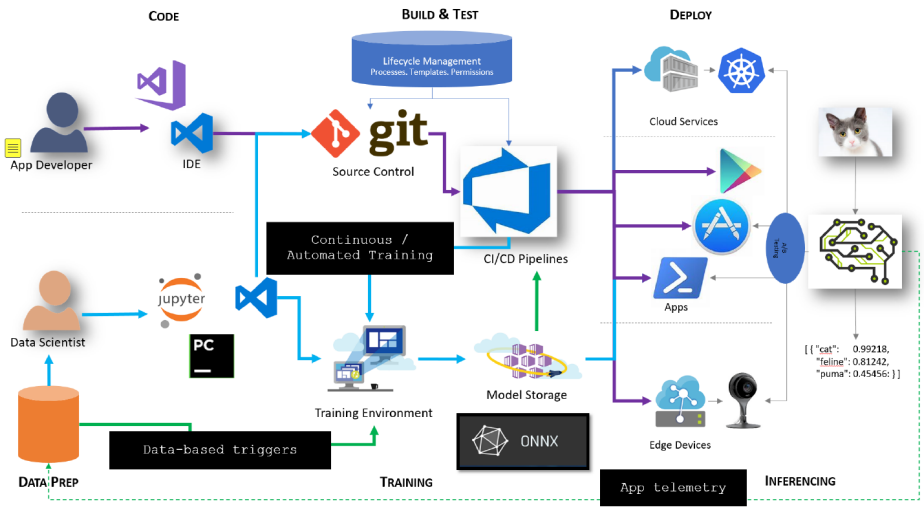

Figure 2: A macroscopic view of where a CI/CD service stands in modern ML application development lifecycle.

and GitHub Actions. We have released an open-source version on GitHub [7] and we have ongoing efforts conducted with Microsoft internal partners to integrate our service into Azure ML Services. In this paper, we present the design and implementation details of our CI service.

1.1 Production Requirements

Making a theoretical framework like `ease.ml/ci` into an industrial-strength tool requires us to revisit the requirements of a real production environment. The first step in our work is to define such requirements from our experience in building real-world ML platforms and applications.

ML Life Cycle. An ML application development lifecycle typically involves multiple iterations. In each iteration, developers try to come up with an ML model with the best quality (e.g., prediction accuracy) on the *training* or the *validation* dataset. This model is then evaluated against a holdout, *test* dataset that is drawn independently from the data distribution that governs the underlying data generation. Based on the quality of the model observed on the test dataset, developers then perform error analysis (if possible) and enter the next iteration. This iterative procedure ends whenever developers are satisfied with the model quality (over the test dataset) or the model quality cannot be improved any more.

CI/CD for ML. A CI/CD solution for iterative ML model development requires supporting numerous types of data sources, a variety of training tools, a validation solution to analyze and validate models (for functionality and performance) and supporting deployment to the infrastructure used to serve models in production. This becomes particularly challenging when data changes over time and fresh models need to be produced regularly, as is the case in many large-scale, AI infused systems. Complexity only grows as models need to be deployed to a hybrid of the Intelligent Edge + Intelligent Cloud. Figure 2 provides an overview of where a "CI for ML" service stands in the whole ML application development lifecycle.

Our Ultimate Goal. Our primary goals here are to (1) standardize the components leveraged for model lifecycle management – model training, model validation, model storage/versioning, model and health monitoring; (2) provide lightweight process and templates to simplify the data scientist/app developer collaboration; (3) reduce the time from model creation to production deployment from the order of months to weeks to days.

1.2 Theoretical Foundation

Recent progress on *adaptive analysis* [13] reveals the fact that the fidelity of the test dataset may fade away when it being accessed again and again. The intuition behind this observation is that, developers may be able to gain insights upon seeing the test accuracy, and then customize their next version of the ML model towards improving the accuracy on this particular test dataset. This obviously may result in *overfitting* — while the prediction error over the given test dataset seems small, the generalization error (which can be estimated using an independently generated test dataset) can be potentially large.

One obvious solution to this problem is to draw an independent test dataset each time when a new version of model is developed. However, this will result in significant overhead in terms of *sample complexity*, i.e., the amount of test data being required. This poses a problem because test data (especially labelled data) is not cheap to obtain. Fortunately, as was shown in [11, 20], it is possible to avoid paying that price as the sample complexity can be reduced to the level that is feasible in practice. This lays the theoretical foundation of developing CI systems for ML applications.

1.3 Data Management

The risk of overfitting due to adaptive analysis requires refreshing the test dataset as CI proceeds, which leads to natural data management problems in terms of effectively maintaining the test data. In addition to standard database operations such as insertion, update, and deletion, users of the CI service may also want to query historical data as well as telemetry information about performance of models that have been submitted in the past. Moreover, users may even wish to keep track of the entire development history and "roll back" to any point in the development trace to "restart"

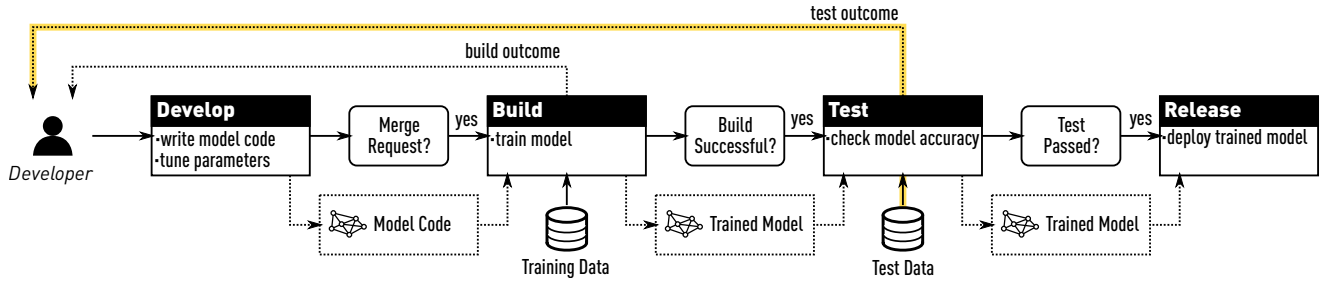

Figure 3: The development lifecycle of a machine learning model in the framework of traditional software development. The shaded yellow line depicts the information leakage pathway that our method tries to solve.

from there. All these requirements need careful design of the data management layer of the CI service to incorporate data versioning and version control mechanisms.

1.4 Paper Organization

We start by presenting the design of the core component, the `mltest` tool, of our "CI for ML" service in Section 2. We then present its implementation details in Section 3. We discuss the new challenges raised by adaptive analysis and our solutions in Section 4. In Section 5 we further present evaluation results that showcase both the necessity and effectiveness of our solutions. We summarize related work in Section 6 and conclude the paper in Section 7.

2 SYSTEM DESIGN

In this section we present an overview of the CI system we have developed, including the interaction model, key design considerations, as well as a walkthrough over the major components it contains.

2.1 An Overview of ML Development Lifecycle

Figure 3 presents an overview of the ML development lifecycle under our CI system. Like the development of regular software, the entire lifecycle consists of four stages (akin to a GitHub or Azure DevOps kind of development scenario):

- *Develop* – the developer writes code for featurizing data, selecting an appropriate algorithm with efficient implementation, as well as basic parameter tuning; the entire ML-related software artifact produced by this stage (including the feature extraction code) is what we refer to as a *model*.
- *Build* – the developer requests merging the code into the master branch (a.k.a., a *pull request*); this automatically triggers the build process of the codebase, which trains the new model over the training data.
- *Test* – the test phase follows if the build process succeeds; the final model returned is evualuated against the test dataset, after which the test accuracy is reported to the developer.
- *Release* – if all test cases are passed and the developer is satisfied with the test accuracy, the model can then be promoted to a release environment for upstream consumption, potentially replacing an old model that was already released.

The key difference of our "CI for ML" system from a classic CI system lies in the *Test* stage:

- *Probabilistic evaluation of test conditions* – unlike test conditions in regular software test that have outcomes which are deterministic, test conditions in our CI system are probabilistic in terms of their semantics.
- *Automatic refresh of test dataset* – whenever the test dataset loses its power as a representative of the underlying data distribution due to repeated accesses, a new test dataset is generated and an automatic swap occurs behind the scenes.

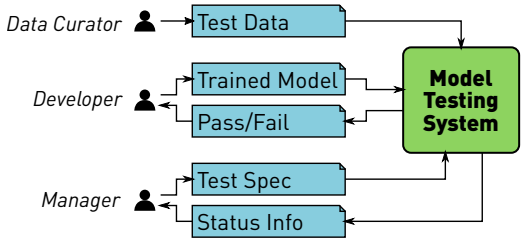

Figure 4: Roles and their interactions with our system.

We will discuss these two respects in more detail soon. Before that, below we give more details regarding the interaction model between our system and the developer.

2.2 Interaction Model

In order to justify various design choices we made, it is important to review the user-facing interface of the system. We define three different types of users (or *roles*). Figure 4 presents the interactions between different roles and our system.

The **manager** role is in charge of defining test conditions. The manager is aware of the broader architecture and all components that make up the user's system, only one of which may be the ML model itself. She is also aware of all tests that have been developed, which gives her the ability to determine which quality standards the model needs to fulfil in order for the overall system to function correctly. Her main point of interfacing with our CI system is the *test spec*. This enables her to control all aspects of model testing, the most important of which is the ***test condition*** that determines whether a new model that was committed will be accepted or rejected. Moreover, the manager also has access to all monitoring tools provided by our system. Mainly, the manager is able to monitor the amount of available test data and the number of test runs that the system can perform before a new test set would need to get staged.

The **data curator** role is in charge of providing fresh test data to the system. The data curator may perform various data preprocessing steps before ***depositing the test data***, which is her main point

of interfacing with our system. Depending on the original source of data and the storage medium, the data curator might benefit from some integration capabilities such as being able to pull data directly from a SQL Database. To achieve this, the data curator may find it useful to implement custom data acquisition adapters.

The **developer** role is in charge of building and improving ML models. The developer may write model code, train the model, and tune its hyperparameters, either manually or by using existing tools and overseeing the process. Most importantly, the developer is in charge of submitting new ML models to our system for *testing* and, potentially in the end, for deployment. Each model that the developer submits triggers a ***test run*** that determines if the model passes the test conditions defined by the manager and is ready for deployment. Depending on the way that our system is configured, the developer may get informed of test outcomes in the form of a binary `pass`/`fail` signal (with built-in probabilistic semantics that we shall discuss later). This signal, albeit a necessary element of the development lifecycle, is the main source of information leakage that our system is trying to control.

2.3 Interfaces

One of the most important design goals of our system is to be easily integrable with as many existing systems and engineering practices as possible. We recognize three prominent interfacing methods that would cover the needs of the vast majority of our users: (1) interacting directly through the command line (CLI), (2) serving Web-based requests (REST), as well as (3) integrating with custom testing code. We implement our solution in Python with support for all three mentioned interfacing methods.

In the following, we present individual interfaces for various functionalities required by all three roles defined in Section 2.2.

2.3.1 Data Management. As we described earlier, the test dataset cannot remain static over the long-term development lifecycle, so it has to be managed as part of the test workflow. Our system achieves this by maintaining a pool of fresh test data where new data can be asynchronously ***deposited*** as soon as it becomes available. New data is added by invoking the `deposit` command.

A subset of that test data gets ***staged***. This is a separate pool of data that is ready to be used for immediate test runs. Each staged dataset has a unique *stage key*. Any staged dataset can be ***loaded*** with the `load` command by using the stage key. If the stage key is omitted, then the latest staged dataset is retrieved.

As our system keeps counting the number of test runs performed over a staged pool of test data, it is able to determine when the maximum allowed number of test runs has been reached (ref. Section 4.1). Once this occurs, the current staged test pool has to be set aside and a new test dataset has to be staged. This is most commonly done automatically, but it is also possible to stage a new test dataset by running a `stage` command.

There is currently no universally accepted interface or storage format for managing ML datasets. What ML datasets have in common is that they are a set of *independent and identically distributed* (i.i.d.) data examples that can be treated individually. In the supervised learning setting that our system focuses on, there is also the concept of *feature* as the input to an ML model and the concept of *label* as its target output. In principle, these concepts are the bare minimum that a data access interface needs to support in order to work with our system.

Our implementation represents datasets by using the Pandas `DataFrame` abstraction, a very popular choice among data scientists. It represents data in tabular form where named columns hold values of the same type. Individual data examples are represented by rows of this table. We expect there to be a single column for target labels and one or more columns for input features, all of which can be specified by name.

2.3.2 Test Condition Specification. As described in Section 2.2, test conditions are used by the manager to define a predicate that needs to be satisfied in order for a model to *pass* the test. This is a necessary step in order to guarantee a certain quality standard of ML models. In typical ML settings, the quality of a model is tested by invoking a *scoring function* over the predictions that the model generated by taking input features from a test dataset. Our implementation currently only supports classification tasks and the model *accuracy* as a scoring function that returns the proportion of labels that the model predicted correctly, as judged by their equality to the true labels that are part of the test dataset.[1]

The result of a scoring function is a single real number from the $[0, 1]$ interval that we call a ***score***. Since any test set is randomly sampled from a (theoretically infinite) pool of test data, the score that we measure is a *random variable*. A test condition is composed of *test clauses*. Test clauses are specified as *inequalities* defined over the ***score*** variables. Each clause is also associated with a *confidence interval* ϵ that enables probabilistic treatment of those clauses. The following is an example test clause:

```
n - o > 0.01 +/- 0.005.
```

Here, the variable `n` represents the score of a newly submitted model that we are currently testing, and `o` is the score of the last model that got accepted by the system. On the right-hand side of the `>` symbol, we have a comparison with a constant `0.01` and a confidence interval `0.005`. Our sample-size estimation method (described in Section 4) ensures that we have enough samples to control the variance of all random variables and correctly evaluate a clause with (high) probability $1-\delta$, where $0 < \delta < 1$ is configurable.

Test conditions represent a conjunction of one or more test clauses. An example test condition made up of two test clauses is:

```
n - o > 0.01 +/- 0.005 and d < 0.01 +/- 0.005.
```

Here, the variable `d` represents the fraction of predicted labels that are different between the newly submitted model currently being tested and the latest model that got accepted.

2.3.3 Test Run. We define a single `run` command that internally orchestrates other system components in order to run a submitted model and evaluate it against the specified test conditions. The command dynamically assembles other components of the system and exposes their configuration options. All options are assigned with sensible defaults, which can be overridden by providing a key-value based configuration file that maps assigned values to configuration options referenced by their keys. Among other things, these options permit specifying user-defined test conditions as well as invoking the `predict` method of the submitted model. Here is

[1] https://en.wikipedia.org/wiki/Precision_and_recall

an example of using the `run` command (we have named our system the `mltest` tool internally):

```
mltest run \
  --config mltest.yml \
  --condition-statement \
    "n - o > 0.01 +/- 0.005" \
  --model-run-command \
   "./predict --data-in {{input}} --data-out {{output}}"
```

It explicitly specifies a test condition and the command used to generate predictions from the trained model. The `{{input}}` and `{{output}}` placeholders are dynamically replaced at runtime by locations of the test data and the temporary output directory, both of which are managed internally by our system.

3 IMPLEMENTATION

The key functional elements the `mltest` tool offers are: (1) compute the number of required test runs given a test condition, and (2) evaluate those test conditions given accuracy score estimates. However, in order to achieve better usability of the system, as showcased in Section 2.3.3 we extend this very narrow range of capabilities by enabling the running of a test in a single-line command, with everything else taken care of transparently. The broader functionality of the `mltest` tool revolves around managing access to data and models, keeping track of all test runs, running models, and estimating accuracy scores. We now present the implementation details of the `mltest` tool.

3.1 Architecture

Figure 5 presents an architectural overview of the `mltest` tool. The `TestRunner` class implements the `run` command and thus hosts the main loop of the system. We define three other interfaces that host major functional components: `DataSource`, `RunLogger`, and `ModelRunner`. This design enables us to host states that represent data, run logs, and models in various locations. Adding a new storage endpoint (for any of these) therefore comes down to providing a concrete implementation of the corresponding interface. Given this separation of interfaces, we are able to mix and match various storage endpoints. For each of these interfaces, we also provide one default implementation that uses `git` as an endpoint for storing data, models, and logs. Git is a convenient system for storing the state because it enables the state to live and evolve together with the main code. Each `git`-based endpoint stores its entire state in a separate branch on the same git repository. Even though (at least a portion of) this state should be inaccessible to the developer, we assume the developers are well-behaving and will not peek into the hidden branch. If extra security is required, the content of this branch could also be encrypted. To increase the modularity and adaptability of our system, we also define the (test) `Condition` as an interface to enable modifying the way of processing test conditions.

3.2 Components

In this section we walk through the four main components of the `mltest` tool mentioned above and discuss the most interesting details (of the default implementations).

3.2.1 Data Source. As mentioned in Section 2.3, our assumptions about a dataset include it being a collection of i.i.d. data examples and that each data example contains one or more *features* and a single target *label*. For simplicity, we restrict ourselves to tabular data and we use Pandas `DataFrame` as a data interface.

We define the `DataSource` interface to provide an abstraction over an arbitrary endpoint that hosts the datasets we use for testing. This endpoint needs to provide the ability for the pool of data to incrementally grow in size by depositing new test examples, to prepare (or stage) a subset of that pool for testing and to load the staged subset. Our current implementation assumes the schema of the data remains constant over time. We expect this to be sufficient for a lot of scenarios because it is still possible to add new features to the model by including them in the feature extraction code. We define the following operations in this interface:

- `deposit` – Feeds the test data pool by adding one incremental batch of data to it. We expect this method to be invoked each time the **data curator** prepares new test data. All deposited data becomes available for staging.
- `stage` – Removes a batch of data examples from the test data pool and produces from it a new ***staged dataset*** that can be used for running tests. All stages have unique keys and represent distinct sets of data examples. The `stage` operation takes an optional argument *size* that specifies the maximal number of data examples taken from the pool of deposited data to form a new stage. If omitted, all available unstaged data will be added to the new stage. A stage can be loaded by using its unique stage key.
- `load` – Returns a staged dataset identified by a stage key. If the stage key is omitted, the last staged dataset is returned.
- `get_size, get_staged_size` – Returns the number of available examples in the unstaged data pool, and the number of examples in a stage identified by the stage key (or the latest stage if the key is omitted).
- `get_keys` – Returns the keys of all stages ever created on a given data source, in chronological order.

We provide a `git`-based implementation of the `DataSource` interface, named `GitDataSource`. It assumes that all test data is stored on an arbitrary branch of a `git` repository. By default this would be a separate branch on the main repository that hosts all other code, but this is configurable. To store large files, we can configure it with `git` Large File Storage (LFS) [4]. The `git` data source keeps all data under a single directory, and creates one subdirectory *per stage* where the name corresponds to the stage key. The data files are stored as JSON-serialized Pandas `DataFrame` instances.

3.2.2 Model Runner. Our system works with trained models that take input features and predict corresponding target labels. They are assumed to be able to contain additional feature extraction components that preprocess the input features before feeding them into an actual trainable model. These models can be hosted on an arbitrary endpoint, which is why we use the `ModelRunner` interface to abstract them away and expose a minimal interface.

A model runner exposes a collection of models, each of which can be identified by a unique model *key*. In the context of CI, each different version of a model will have a unique key. We assume that all versions belong to the same ML task and have been trained on datasets following the same underlying data distribution. We define the following operations in the `ModelRunner` interface:

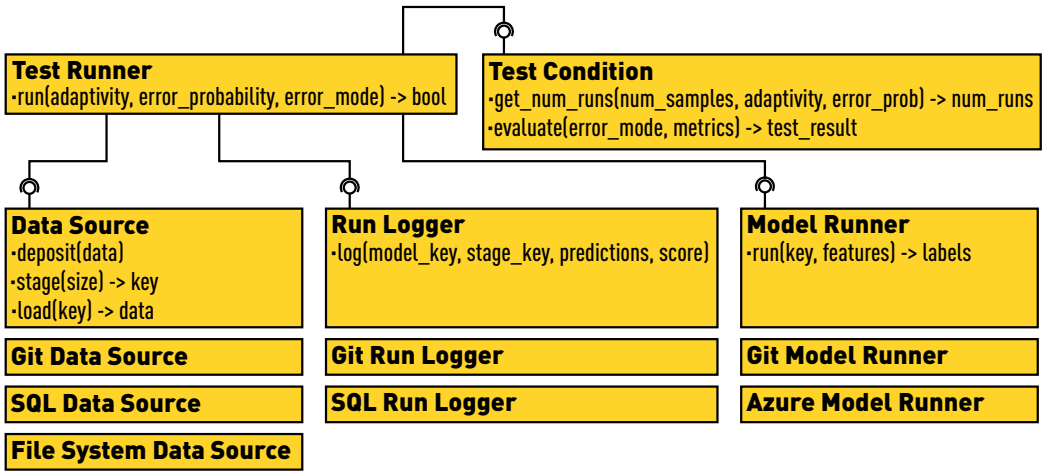

Figure 5: An architectural overview of the implementation of the `mltest` tool.

- `run` – Invokes the `predict` function of a model identified by the key. If the key is omitted, the latest version of the model is targeted. We pass the features of a batch of data examples and the targeted model returns the predicted labels.
- `get_keys` – Returns the keys of all (versions of) models that are available on a given model storage endpoint, in chronological order.

We again provide a `git`-based implementation of the `ModelRunner` interface, named `GitModelRunner`. It assumes that all models are committed to an arbitrary branch of a single `git` repository. The model keys thus correspond to the hashes of all commits that contain updates to the model — we want to avoid treating updates to non-model files as new model versions. To run a model with a specific key, the `git` model runner checks out the commit based on the hash and invokes its `predict` command. This command is an input parameter of the `git` model runner.

3.2.3 Run Logger. We use the `RunLogger` interface to maintain a unified record of all test runs that took place for a given data source and a given model runner. A test run is implemented by the `Run` class which has a unique *run key*, as well as the corresponding *model key* and *stage key*. It also holds all predictions generated by a model, the accuracy score of the model, and the test outcome which is a Boolean `pass/fail` indicator.

The pool of test runs can be queried by model key and/or by stage key. The query can either return serialized run descriptors or simply return the count of runs that satisfy a predicate. We define the following operations in the `RunLogger` interface:

- `log` – Submits a new run to the log. This method accepts the targeted model key and stage key, as well as the predictions, the score, and the test outcome of the corresponding model. Once a run is created, its run key is returned. Runs can be queried either individually or collectively.
- `get_run` – Returns a run identified by its run key.
- `get_runs, count_runs` – Queries the whole pool of runs by using a specific model key or stage key, or both, as a search criterion. The `get_runs` operation returns the run instances, whereas the `count_runs` simply counts the number of runs satisfying the search criterion.

The most important operation is `count_runs` that returns the number of runs for a given stage, because it enables our system to impose limits on test runs over a single test dataset.

Once again we provide an implementation of the `RunLogger` interface with a `git`-based approach, named `GitRunLogger`. Just like the `GitDataSource` implementation, it stores all runs in some branch of a specified `git` repository. By default, this would be a separate branch of the `git` repository that hosts the model. Serialized runs are stored as JSON files in a specified directory.

3.2.4 Test Condition. We implement the `Condition` class in order to encapsulate the functionality required to work with test conditions. We need to be able to parse them from a string representation, use them to compute the number of permitted test runs for a given number of samples, as well as use accuracy estimates that are treated as random variables and evaluate a test condition to compute a `pass/fail` outcome.

Following `ease.ml/ci`, we define a *domain specific language* (DSL) that allows expressing test conditions in a compact way. The DSL is simple but is able to encode a large number of test clauses that are interesting in practice. The top-level literal of the DSL is a ***condition***, which is simply a conjunction of clauses:

```
condition := clause "and" condition | clause.
```

The building blocks of each condition is a ***clause*** that is a simple inequality with an expression on the left-hand side and a constant real value on the right-hand side, along with an error margin ϵ:

```
clause := expression (">"|"<") constant "+/-" constant.
```

Here, an ***expression*** is a summation of factors:

```
expression := factor ("+"|"-") expression | factor,
```

where each ***factor*** is made up of a variable multiplied by one or more constants:

```
factor := constant "*" factor | variable.
```

Finally, a ***constant*** can be any real value, and a ***variable*** is one of:

- `n` – the accuracy of the *new* model, i.e., the newly submitted model that we are currently testing;

- `o` – the accuracy of the *old* model, i.e., the previously submitted model that we last tested before the new one;
- `d` – the fractions of predictions that are different between the *new* and the *old* model, used to control model stability.

After we parse a test condition expressed using the above DSL, we are able to perform several useful operations on it:

- `evaluate` – Computes a Boolean `pass`/`fail` outcome of a test condition given the estimated values of the variables, in terms of an *error mode* that defines how to deal with type I and type II statistical errors (details in Section 4.2).
- `get_num_runs` – Computes the *number of test runs* permitted on the (staged) test dataset in order to protect it from overfitting, given an *error probability* that defines the plausibility of the test outcome (since the outcome itself is a random variable), an *adaptivity mode* that prescribes the regime by which information will be released to the user, and the *number of samples* we have in a (staged) test set (details in Section 4.1).
- `get_num_samples` – Works similarly as `get_num_runs`, with the only difference that it returns the *number of samples* needed to support a given *number of test runs* instead of the other way around.

One of our goals is to enable users to get instant feedback while trying out different settings for their test conditions. For this purpose, all functionalities of the `Condition` class are exposed through all three interfacing methods defined in Section 2.3.

4 EVALUATION OF TEST CONDITIONS

We present details of the evaluation of test conditions in this section. Specifically, we provide solutions to the following two problems:

- *Test run estimation* – given the size of the test dataset, estimate the number of runs/submissions that it can support;
- *Probabilistic semantics* – what kind of probabilistic guarantees we can provide for the evaluation outcomes.

We note that the theoretical foundation of the techniques we present here has already been laid out by Renggli et al. [20]. We therefore focus on their implementation in the `mltest` tool.

4.1 The Number of Test Runs

As shown in [20], the number of test examples required for an (ϵ, δ)-guarantee for a single test condition (e.g., that makes an assertion of the model accuracy) is:

$$n(v, \epsilon, \delta) = \frac{-\ln \delta}{2\epsilon^2}. \tag{1}$$

We use v to represent the random variable we are estimating, e.g., the model accuracy. If we have the number of test examples indicated by Equation 1, we can then guarantee that $\mathbb{P}(|\hat{v} - v| \geq \epsilon) \leq \exp\left(-2n\epsilon^2\right) \leq \delta$, where $\hat{v}$ is the estimated value of v.

To support more complex test conditions, however, we need to deal with expressions elaborated in Section 3.2.4. The simplest one is multiplication with a constant:

$$n(c \cdot v, \epsilon, \delta) = n(v, \epsilon/c, \delta). \tag{2}$$

For a summation or difference we have:

$$\begin{aligned} n(v_1 \pm v_2, \epsilon, \delta) = &\max\{n(v_1, \epsilon_1, \delta/2), n(v_2, \epsilon_2, \delta/2)\} \\ &\text{s.t. } \epsilon = \epsilon_1 + \epsilon_2. \end{aligned} \tag{3}$$

In both cases we need to solve an optimization problem with the constraint $\epsilon = \epsilon_1 + \epsilon_2$. This permits us to construct arbitrary clauses defined in Section 3.2. Finally, we want to handle a test condition that is a conjunction over multiple clauses C_1, ..., C_k. The number of samples we need for the conjunction is equal to the number of samples to evaluate the hardest clause in the conjunction:

$$n(C_1 \wedge \ldots \wedge C_k, \epsilon, \delta) = \max_i n(C_i, \epsilon, \delta/k). \tag{4}$$

4.1.1 Adaptivity Modes. So far, we have been focusing on estimating the number of samples for a single test run. If we wish to use the same test dataset to support multiple test runs, then it may lose its statistical power (as a representative of the true data distribution) due to the presence of adaptivity. In our system, we provide two *adaptivity mode*s: the **non-adaptive** mode and the **full-adaptive** mode. The number of test runs that can be supported depends on the mode our system operates in.

***Non-adaptive** Mode.* In this mode, no information is ever revealed, i.e., the outcome of the test is completely hidden. This mode is rarely useful in practice based on conversation with our partners. Nonetheless, it serves as a special (and the unique) case where we can safely treat all submitted models as *independent*, which significantly increases the number of test runs (supported by a certain test data size). Suppose that we want to run N independent models over a single staged test set. By the union bound, it follows that

$$\delta = \bigcup_{i=1}^{N} \delta_i \leq \sum_{i=1}^{N} \exp\left(-2n\epsilon^2\right) = N \cdot \exp\left(-2n\epsilon^2\right). \tag{5}$$

This means that the required test dataset size for N models is:

$$K = n(v, \epsilon, \delta/N) = \frac{-\ln(\delta/N)}{2\epsilon^2}. \tag{6}$$

For a given K it thus follows that $N = \delta e^{K \cdot 2\epsilon^2}$.

***Fully-adaptive** Mode.* In this mode, after every evaluation the test outcome is released, which is typical in CI scenarios. By Figure 3, such information leakage creates a dependency between models submitted over the same staged test set. As a result, the union bound and thus the test set size derived in Equation 6 do not apply.

As was shown in [20], by using an idea similar to the Ladder mechanism [11], we can, however, apply the union bound over all 2^N possible (binary) sequences that represent the whole space of possible test outcomes from N submitted models:

$$\delta = \bigcup_{i=1}^{2^N} \delta_i \leq \sum_{i=1}^{2^N} \exp\left(-2n\epsilon^2\right) = 2^N \cdot \exp\left(-2n\epsilon^2\right). \tag{7}$$

As a result, the required test data size is

$$K = n\left(v, \epsilon, \delta/2^N\right) = \frac{-\ln(\delta/2^N)}{2\epsilon^2}, \tag{8}$$

which gives $N = \frac{\ln \delta + K \cdot 2\epsilon^2}{\ln 2}$.

4.2 Probabilistic Semantics

The major functionality of test conditions is to produce a reliable `pass`/`fail` signal. In our context, all values that we pass to our three variables `n`, `o`, and `d` are random variables, which implies that the evaluation outcomes should also be treated as random variables. How do we, then, interpret the semantics (i.e., probabilistic guarantees) of the evaluation outcomes?

Consider, for example, the test condition `n > 0.6 +/- 0.05`. If we measured `n` to be 0.7, should this clause be evaluated as `true`? The answer is yes, but only if we used at least the amount of samples prescribed by Equation 1. That bound provides us with a statistical guarantee that, with probability δ, our estimate of `n` will *not* be more than ϵ-away from the true `n`. Since in this case $\epsilon = 0.05$, it is safe to evaluate this clause as `true`.

Now, consider the case if we measured `n` to be 0.61. Since this is *within* ϵ-distance from 0.6, we cannot rely on our statistical guarantees. When these situations occur, it is not possible to avoid sometimes returning a wrong result and it is not possible to know when the result is wrong. In such cases, the best we can do is to consider a trade-off and choose between false positives and false-negatives. In other words, we can choose to either be *false positive free* and always treat these unknown situations by returning `false`, or be *false negative free* and always return `true`. Accordingly, we further provide two *error mode* settings in the `mltest` tool, which can be either `fp-free` or `fn-free`.

5 EVALUATION

We try to answer two main questions in our evaluation:

(1) What is the danger of using a static test set in CI and how can our method help?
(2) Given a test condition and our methods presented in Section 4, how many test runs can be performed against a single staged test set, with what guarantees and at what price?

5.1 Battle Against Overfitting

We simulate a *develop-commit-test* scenario in this experiment. We set up a test system that accepts a new model only if its estimated test score is greater than the current best model. We take a real dataset that represents a classification task regarding the presence of Higgs bosons in a physical process [10]. We split this dataset into a training set (with 20k data examples), a running test set of size N that is used by our test system, and a large test set with *one million* data examples that we use to estimate the "true test score." Each time the developer submits a model, we run a test procedure and send the `pass/fail` outcome back to the developer.

We simulate the developer's submission activities as a random process that first randomly picks a model from a set of available classifiers and then randomly picks hyperparameter values from predefined ranges. The set of candidate classifiers includes *random forest*, *extra trees*, *decision tree*, *k-nearest neighbors*, and *linear SVM*.

As a baseline approach for comparison, we simulate a scenario where the dataset used for testing remains unchanged over the entire duration of a development lifecycle. We use a static test set of size 500. As depicted in Figure 6, it becomes clear that, as time goes by, the estimated test score begins to *diverge* slowly from the true test score. We thus conclude that, if we use a static test set in a "CI for ML" system, in certain scenarios we may end up in a situation where developers can overfit the test set even though only a single bit of information is disclosed per test run.

In Figure 7 we examine how our approach manages to deal with the overfitting problem. The test condition and the development cycle remain the same. The only difference is that now we use our stage-based method for estimating test scores. We conduct three

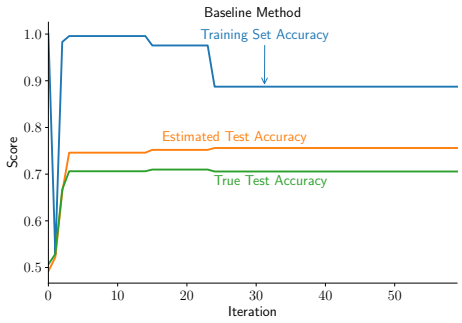

Figure 6: A scenario where the developer overfits the test set.

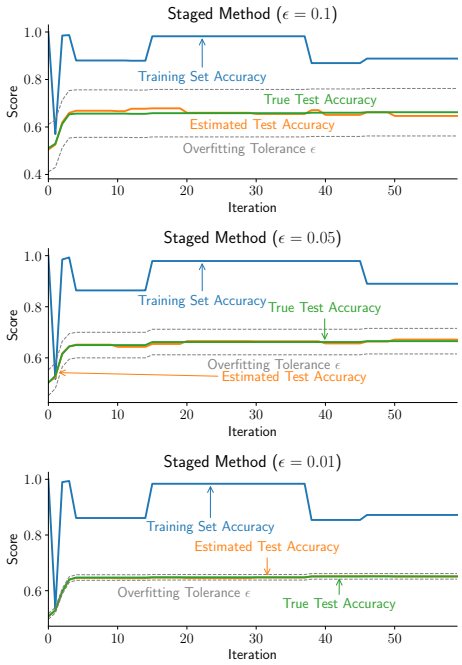

Figure 7: Our strategy ensures that the estimated test error will be within the ϵ-distance of the true test error.

experimental runs, each time ensuring that the estimated test score remains within ϵ-distance from the true test score (with probability $1 - \delta = 0.99$). We try out three values of ϵ: 0.1, 0.05, and 0.01. Since we run 10 tests for each staged test set, by Equation 8 the staged test sets need to contain 577, 2307, and 57683 data examples respectively, under the ***full-adaptive*** mode. It is easy to observe that the estimates do indeed always stay within bounds of our ϵ-margin (shown by the dotted lines).

5.2 The Cost of Running Staged Tests

We present an evaluation of our method with the goal of gaining intuition about the cost of running staged tests. We measure how many runs can be performed on a test set of a given size with required quality guarantees.

We study the impact of the input parameters to the method we use for computing the number of runs (described in 4.1). We try out three test set sizes with 50k, 100k, and 500k data examples. We try two representative test condition categories: One that checks if

	test condition:		n - o > 0.1 +/- 0.1	n > 0.5 +/- 0.1
test set size	ϵ	δ	# of runs	# of runs
$50k$	0.01	0.0001	n/a	2 ($208)
		0.001	n/a	5 ($83)
		0.01	n/a	8 ($52)
	0.025	0.0001	8 ($52)	76 ($5)
		0.001	11 ($37)	80 ($5)
		0.01	14 ($30)	84 ($5)
$100k$	0.01	0.0001	n/a	16 ($46)
		0.001	n/a	18 ($36)
		0.01	n/a	23 ($5)
	0.025	0.0001	30 ($28)	168 ($5)
		0.001	34 ($24)	170 ($5)
		0.01	38 ($22)	174 ($5)
$500k$	0.01	0.0001	21 ($198)	130 ($32)
		0.001	25 ($166)	134 ($31)
		0.01	29 ($143)	138 ($30)
	0.025	0.0001	211 ($20)	889 ($5)
		0.001	215 ($19)	891 ($5)
		0.01	217 ($19)	895 ($5)

Table 1: Number of runs and (in braces) a price estimate for a single run for various test set sizes, and (ϵ, δ) values. Cells marked as n/a correspond to insufficient test set sizes.

the test score is above a certain threshold, and another that checks if there is an improvement in the test score when comparing two models. For each test condition, we try out different combinations of the error margin ϵ (taken to be either 0.025 or 0.01) and the error probability δ (taken to be either 1%, 0.1%, or 0.01%).

We associate each test run with a hypothetical "dollar price" as follows. We assume a labelling effort that is conducted at the speed of 5 seconds per label, and at the price of $6 per hour (number taken from `labellbox.com`). At this rate, 720 data examples can be labelled per hour. We do not include the price of acquiring features, though it is not negligible. We argue that the overall price is dominated by the labelling cost since each data example requires human effort. With this setting, we can then compute the price of a test set (and thus the test run). Table 1 presents the results.

6 RELATED WORK

CI has been an industrial standard in practice and the literature on classic CI in software engineering is overwhelming [12]. However, so far little work has been done towards "CI for ML," although there have been emerging discussions in online communities regarding such requirements [8, 17, 18, 26]. The recent effort by Renggli et al. [20, 21] is the first work along this line, as far as we know. It lays out theoretical foundations as well as building a proof-of-concept system to demonstrate the feasibility of "CI for ML." The current paper, meanwhile, takes one step further by addressing the design, implementation, data management, integration challenges for developing an industrial-strength "CI for ML" service.

The "CI for ML" idea also fits well into the broader scope of building AutoML systems and services. Users of AutoML systems only need to provide their data and high-level specifications of their ML tasks (e.g., loss functions to be minimized), and the system will take over the rest of the job, such as automatic pipeline execution, resource allocation, and performance monitoring. Typical AutoML systems include industrial offerings from major cloud service providers such as Amazon SageMaker [1], Microsoft Azure Machine Learning [6], and Google Cloud AutoML [5], as well as ones from academic institutions such as the Northstar system developed at MIT [16], and the `ease.ml` service [15, 19, 27] by ETH Zurich, among others. It remains interesting to see how to incorporate "CI for ML" into these existing AutoML services.

7 CONCLUSION

We have presented our efforts and experiences with building an industrial-strength "CI for ML" service. We discussed the details of its design, implementation, data management, and integration, as well as evaluation over real datasets. We showcased the risk of overfitting a static test set in the context of "CI for ML" that motivated us to come up with the "staged test set" solution, and demonstrated its affordable cost in practice.

REFERENCES

[1] Amazon sage maker. https://aws.amazon.com/sagemaker/.
[2] Aws codepipeline. https://aws.amazon.com/codepipeline/.
[3] Azure devops services. https://azure.microsoft.com/en-us/services/devops/.
[4] Git LFS. https://git-lfs.github.com/.
[5] Google cloud automl. https://cloud.google.com/automl/.
[6] Microsoft azure machine learning. https://azure.microsoft.com/en-us/services/machine-learning/.
[7] `mltest` tool open-source repository on github. https://aka.ms/gsl-ml-test.
[8] Continuous integration for machine learning. https://medium.com/@rstojnic/continuous-integration-for-machine-learning-6893aa867002, April 2018.
[9] S. Ackermann et al. Using transfer learning to detect galaxy mergers. *MNRAS*, 2018.
[10] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.
[11] A. Blum and M. Hardt. The ladder: A reliable leaderboard for machine learning competitions. In *ICML*, pages 1006–1014, 2015.
[12] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[13] C. Dwork et al. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015.
[14] I. Girardi et al. Patient risk assessment and warning symptom detection using deep attention-based neural networks. *LOUHI*, 2018.
[15] B. Karlas, J. Liu, W. Wu, and C. Zhang. Ease.ml in action: Towards multi-tenant declarative learning services. *PVLDB*, 11(12):2054–2057, 2018.
[16] T. Kraska. Northstar: An interactive data science system. *PVLDB*, 11(12):2150–2164, 2018.
[17] A. F. Lara. Continuous integration for ml projects. https://medium.com/onfido-tech/continuous-integration-for-ml-projects-e11bc1a4d34f, October 2017.
[18] A. F. Lara. Continuous delivery for ml models. https://medium.com/onfido-tech/continuous-delivery-for-ml-models-c1f9283aa971, July 2018.
[19] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *PVLDB*, 11(5):607–620, 2018.
[20] C. Renggli et al. Continuous integration of machine learning models with ease.ml/ci: Towards a rigorous yet practical treatment. In *SysML*, 2019.
[21] C. Renggli, F. A. Hubis, B. Karlas, K. Schawinski, W. Wu, and C. Zhang. Ease.ml/ci and ease.ml/meter in action: Towards data management for statistical generalization. *PVLDB*, 12(12):1962–1965, 2019.
[22] K. Schawinski et al. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. *MNRAS*, 2017.
[23] K. Schawinski et al. Exploring galaxy evolution with generative models. *Astronomy & Astrophysics*, 2018.
[24] D. Stark et al. PSFGAN: a generative adversarial network system for separating quasar point sources and host galaxy light. *MNRAS*, 2018.
[25] M. Su et al. Generative adversarial networks as a tool to recover structural information from cryo-electron microscopy data. *BioRxiv*, 2018.
[26] D. Tran. Continuous integration for data science. http://engineering.pivotal.io/post/continuous-integration-for-data-science/, February 2017.
[27] C. Zhang, W. Wu, and T. Li. An overreaction to the broken machine learning abstraction: The ease.ml vision. In *HILDA@SIGMOD 2017*, pages 3:1–3:6, 2017.