mltest - A testing library for ML projects

Delft University of Technology - Release Engineering for Machine Learning Applications (CS4295)

Group 9

Simran Karnani
4798341

Jord Molhoek
4932919

Ziad Nawar
4765575

Abel Van Steenweghen
4876431

Mirijam Zhang
4660129

Abstract

Testing is essential for the development and maintenance of an accurate and reliable machine learning model. We present the *mltest* library, which offers a set of general ML-specific and configurable tests. The tests in the *mltest* library are inspired by [1]. Finally, we also provide an example machine learning project that integrates continuous testing with a full CI/CD pipeline and that serves a model that can predict tags for Stack-Overflow questions. This system was built from [2].
All code is available on GitHub[1] and Docker Hub[2].

1 Introduction

To ensure that a machine learning (ML) system is accurate and reliable it should be tested continuously. Despite ML being a popular and active field of research, the use of testing frameworks targeted at ML projects is not a common practice. Tests are often limited to unit-testing logic-specific code and monitoring, validating and updating the models and the data they are built upon is often not as streamlined.

We developed a testing library based on the ML test score rubric published by Breck et al. [1] at Google Research. The goal of this library is to make it easy to write tests for areas and scenarios specific to ML projects. The tests are configurable and applicable to general ML projects.

In total, we developed 14 tests covering 4 categories: features and data, model development, ML infrastructure, and monitoring tests for ML. Every test is accompanied by an example implementation, to clarify how it can be used. We attempted to minimize the amount of configuration for the test implementations while maintaining their applicability to general ML projects.

We tested and applied our library on a multilabel classification algorithm for Stack Overflow tags [2]. The project was first segmented into its main components that could independently run. We used DVC to organize the components into a working pipeline. This was further extended by constructing Docker images to independently run the pipeline, providing Kubernetes support, adding data validation, serving the model, and monitoring with Prometheus.

[1] https://github.com/ZiadNawar/remla-project
[2] https://hub.docker.com/r/ziadnawar/remla-project

2 Related Work

With the increase of ML applications, concerns about the trustworthiness of these applications were arising. Compared to the traditional software systems, ML testing is challenging due to the fundamentally different nature and construction of these applications. These models are data-driven and are prone to change in response to new data. However, ML systems can be difficult to test because they are designed to provide answers to a question that has no existing answer. Furthermore, the effects of ML systems can only be understood by considering the entire system as a whole, which makes it difficult to test small units in isolation [4].

Zhang et al. [4] performed a study on ML testing according to four different aspects: the testing properties, ML components, testing workflow, and application scenarios. The workflow describes how to test, the testing properties look at what to test and the testing components are about what to test. These aspects are essential because ML applications may not only have bugs in the code but also in the data that is being used as input to the model. Some key procedures and activities in ML testing are offline testing and online testing. Offline testing tests the model with historical data without a real application environment, whereas online testing aims to detect bugs after the model has been deployed online. Some of the components of ML testing are bug detection in data, in the framework, and in the learning program. The conditions that ML testing needs to guarantee for a trained model are correctness, model relevance, efficiency, robustness, fairness, and interpretability.

Breck et al. [1] also build on the topic of ML testing, where they focus on an ML test score. This work proposes 4 categories of tests, each covering 7 concepts that should be tested in an ML project:

(1) Tests for Features and Data
(2) Tests for Model Development
(3) Tests for ML Infrastructure
(4) Monitoring Tests for ML

Our paper strongly builds upon the paper by Breck et al. [1]. Some proposed tests require access to the environment in which the project is executed, or are dependent on the infrastructure on which the project operates. These tests are omitted from our library, since we want an easily configurable library, that can seamlessly be integrated with a project's current operations.

3 System Design

First, we restructured the example project [2], from a Jupyter Notebook file that processes data and trains and evaluates a model, to a pipeline that is deployed in such a way that it is easy for developers to make changes and such that it can be used by end-users.

Additionally, the mltest library is built and used for this project. The design of this entire system is described in detail in this section.

3.1 Architecture

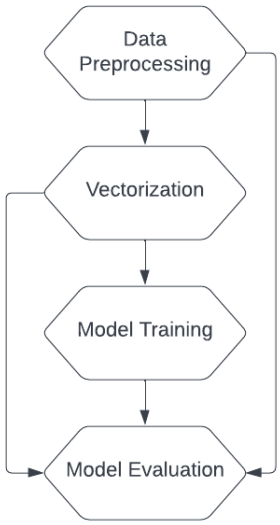

Figure 1: Directed acyclic graph representing the dependencies in the pipeline.

We divided the baseline multilabel classification algorithm into multiple stages, forming a pipeline architecture. Variables are passed between the stages using the joblib library, this allows for separately executing and updating stages of the pipeline. If, for example, the model training procedure has changed slightly, the model training stage and downstream stages need to be re-executed. The stages *before* the changed stage are skipped. This setup can save a lot of time when updating or executing the individual components. The dependencies in the pipeline are visualized in Figure 1. In this figure, the pipeline should be executed from top to bottom. The algorithm stages are discussed in chronological order in the following sections.

3.1.1 Text Preprocessing The input data is loaded and split into proper segments. Next, the data is preprocessed such that all text is lowercase, bad symbols are replaced or removed and stop-words are removed. This makes processing in later stages easier. Next, a dictionary of word counts and a dictionary of tag counts are generated. This is used later for the bag-of-words model.

3.1.2 Vectorization The vectorization stage needs the output of the preprocessing stage. For each entry of the preprocessed data, two high-dimensional embeddings are calculated. One for bag-of-words and one for tf-idf. The purpose of vectorization is to generate a feature vector. In this feature space, similar sentences should be close. Likewise, dissimilar sentences should have a larger distance. This makes learning easier, as this would not be possible on plain text. The vectorization stage outputs the vectorized data to a joblib file.

3.1.3 Model Training Model training takes the vectorized data and fits a logistic regression classifier on both kinds of vectorized data (bag-of-words and tf-idf). As before, these classifiers are stored.

3.1.4 Model Evaluation The model evaluation stage needs the trained models and the vectorized validation data. The models are used to predict the labels of the validation data. These predictions are then compared to the given “true" labels using different metrics. The evaluation results are automatically stored in a report.

3.2 Pipeline Extensions

3.2.1 DVC A simple DVC (Data Version Control) setup is used to manage the architecture as described in section 3.1. The input and output dependencies of the stages in the pipeline (as visualized in Figure 1) are specified in the dvc.yaml file. If a developer wants to run the pipeline, they should do this through “dvc repro". DVC can then observe what files have changed and correspondingly decide what pipeline stages to skip and what stages to run.

3.2.2 Docker Compose We use docker compose to deploy two services: the classifier itself, embedded in a Flask[3] API, and a server to interact with it. For these services we made two images available on the Docker Hub Container Image Library, one for our multilabel classifier[4] and one for the server[5].

3.2.3 Kubernetes For our Kubernetes setup, one YAML file was set up describing two Pod objects, one Service object and one Ingress object. Furthermore, Kubernetes makes use of the published docker images. Each pod runs a container for one of the images. One service is used to target both pods.

3.2.4 Data Validation Data is validated to ensure that what we are expecting from the data is what the model is also seeing. Given a training set, validation set, and test set, the data is validated based on its presence, the length of the titles, its type, the distribution of data (title length and number of tags), and the number of tags. If a developer wants to validate this data, it is done automatically on release using Github Actions.

3.2.5 Model Serving The model is fully implemented in python but some part of our system is implemented in java. The model runs on an independent server and communicates to a java spring boot server with a simple function which has input from the java server the text and the output from the python server is the model’s prediction. The model is then served from the java server which renders a web page to the user. The user can input text and then the server processes it and sends the response to the user on the same web page. An example of a use case in the UI is shown in Figure 2.

[3] https://flask.palletsprojects.com/en/2.1.x/
[4] https://hub.docker.com/r/ziadnawar/remla-project
[5] https://hub.docker.com/r/ziadnawar/springbootserver

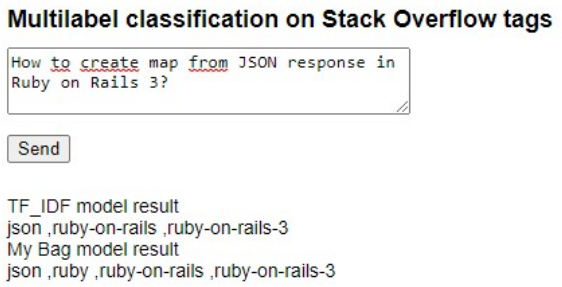

Figure 2: Example text in the simple UI that serves the models.

3.2.6 Monitoring To ensure that the quality of the model is being monitored by the developers, we have developed the model to give textual feedback in a format that we can use. The textual format was the appropriate format for Prometheus would need as input. The monitoring feature is developed in two different aspects. The first part of our system where the server stores data about the results such as the number of requests and the average length of the input. These two metrics were used as a proof of concept. Other metrics would be the model's accuracy by developing an additional input from the user to input the ground truth label or their guess and compare the model results. The second part for monitoring is developed in Prometheus and Grafana where we plot the metrics mentioned to monitor the model's performance.

4 ML Test Library

Our main contribution is *mltest*, a machine learning test library that focuses on having an accurate and reliant ML development pipeline. We based this library on the general test concepts proposed by [1]. The goal of this library is to make implementing these tests in other ML projects simple and effortless. The source code of the test library as well as examples of how the tests can be implemented in an existing code base are available through our GitHub repository [3]. We will now cover the tests our library offers.

4.1 Features and Data

Features and data tests examine the structure of the data on which ML models are built. Data quality and understanding is a fundamental part of producing an accurate ML model, and should therefore be tested as rigorously as normal code. We implemented the following tests.

4.1.1 Pairwise Feature Correlations calculates the correlation of each pair of features and asserts whether none of the features are perfectly correlated. If the correlation is strictly equal to 1, one of the features can be removed. Furthermore, the correlation of each feature with the target is calculated. It is asserted that the correlation is nonzero. If a feature has 0 correlation with the target, it could still be useful in combination with other features. Nevertheless, this is something that might require deeper investigation by the users of the library.

Example use case: a developer has too many features and is wondering which ones they can discard in order to combat the curse of dimensionality.

4.1.2 Unsuitable Features tests that a model does not contain any features that have been manually determined as unsuitable for use. The tester manually provides a list of the names of unsuitable features.
Example use case: exclude gender as a feature for predicting a credit score, and test that the feature is actually excluded.

4.1.3 Feature Generation Validation tests that the code that generates the features is correct. It is important not to overlook this seemingly simple test, as a bug can slip in easily and have detrimental consequences if the model based upon this is used in practice.
Example use case: preprocessing textual data (strings) such that it is all lowercase and it contains no unexpected symbols.

4.1.4 Feature Distribution Expectation tests that the distributions of each feature match your expectations. It can verify that a feature can only attain certain values, but also analyze the frequencies of certain feature values.
Example use case: verify that the word "the" accounts for approximately 5% of the words in a text.

4.2 Model Development

Model development tests target the process of the development of the model. It is wise to have automated tests that repeatedly and critically evaluate the quality and necessity of the model under development.

4.2.1 Compare Against Baseline tests that a model performs better than a simple baseline. The default baseline model is linear, but a logistic classifier can also be selected. It is useful to have this comparison as an automated test, as it is something that can easily be overlooked.
Example use case: repeatedly make sure that the developed model is really necessary. If the accuracy of a complex machine learning architecture is lower than the accuracy of a simple baseline, it is wise to consider how necessary the complex model really is.

4.2.2 Impact of Tunable Hyperparameters uses grid search to find the optimal (hyper)parameters to be fed to the model. Furthermore, the optimal (hyper)parameters are compared to the ones currently used. The test returns the optimal (hyper)parameters and the percentage of non-optimal (hyper)parameters.
Example use case: if data changes over time (as more training data comes in), it may be that the hyperparameters also need to change. This can now be checked automatically.

4.2.3 Quality on Important Data Slices tests the model performance on different slices of data that are segmented along certain dimensions of interest. Examining different data slices avoids having fine-grained performance issues masked by a global summary metric.
Example use case: slicing data by country when determining the heights of different people.

4.2.4 Model Staleness measures the live metrics of the latest model and compares it to those of a previous model. Models should be updated to account for changes in the external world.
Example use case: word associations keep changing over the years. A model based on these associations should be updated to reflect these changes.

4.3 ML infrastructure

A machine learning project is more than just the model. It is important to test all elements of the pipeline, and to make sure that all stages of the pipeline can work together.

4.3.1 Reproducibility of Training trains a model twice on the same data and evaluates on the same test data. Afterwards, it is asserted that the accuracies do not differ more than a given tolerable error. This test can give an indication of how much influence nondeterminism has on the accuracy of the model, and how much variation can be expected.
Example use case: verify that nondeterminism in the model training does not have a huge impact.

4.3.2 Integration Test is technically speaking done by running "dvc repro", but since not every user will be making use of DVC and for completeness of the mltest library, a simple integration test functionality was added. This test simply runs everything in the pipeline to verify that no errors occur when all components come together.
Example use case: repeatedly make sure that all components of the pipeline can work together.

4.3.3 Model Quality tests that the accuracy of the model is higher than the accuracy of a previous model on a predefined test set. This can be useful to make sure that improvements are in fact improvements.
Example use case: developers are notified when an improvement to the model/pipeline turns out to have a negative effect on the accuracy.

4.4 Monitoring Tests for ML

It is important to monitor models that automatically incorporate new data continuously at training time. Monitoring is also needed for models that serve predictions in an on-demand fashion.

4.4.1 Data Invariants tests that all the data of a feature(s) follow certain properties. Some examples include feature A and feature B should have the same number of non-zero values in each example, or feature C should always be in a certain range, or that the class distributions should be about 5:2.
Example use case: the length of a Dutch postal code should always be six characters long.

4.4.2 Training and Serving Features compares the code that generates features for training and for serving the model. These two procedures use slightly different code because training happens on a large amount of data and inference happens on one data point at a time. For a given instance, it should not matter which of these procedures are followed; the representation should be the same.
Example use case: make sure that there is no "training/serving skew".

4.4.3 NaNs or Infinities checks that no invalid numeric values occur during training or in the parameters of the final model. Knowing that these have occurred can speed up the diagnosis of the problem.
Example use case: developer is notified when unexpected values occur, and can debug accordingly. Unexpected numeric values should not go unnoticed.

5 Implications

Machine Learning testing is different from software testing because not only the system needs to be tested, but also the data that the system is being trained on. This extended system is different because it contains the additional ML test library that tests the data at different stages and whether the model works as expected. ML testing also helps the developer get a better understanding of the data, its distributions, attributes, and much more.

The ML Test library is made as a separate library that contains tests for any general model. Therefore, it is not only specific to the model used for this paper but can be extended to other research as well. It contains tests that test whether the data is correct and relevant, whether the model performs well on the data, the entire pipeline of the project, and compares all the features and outputs when new data is continuously being added to the system.

Although it has not yet been implemented as an automatically returned metric, the quality of a machine learning project can be determined with the ML Test score (Section 5.1). This gives developers an indication of how well their project has been tested and based on this score they can determine whether the system is ready to be deployed or requires further improvements. For now, developers are supposed to manually calculate their ML test score, which is not more than 2 minutes of work.

5.1 ML Test score

In [1], based on which the tests in the *mltest* library are inspired, an "ML Test Score" is calculated based on the tests. The score is calculated such that for each test one point is awarded if it runs manually, and two points are awarded if the test runs automatically. Furthermore, the final ML Test Score is the minimum of the scores aggregated for each of the 4 sections described in section 4 [1].

Since the mltest library can easily be used to generate a test suite, and since any test suite can easily run automatically by making use of GitHub workflows or GitLab pipelines, we count two points per test described in section 4. Therefore, if a developer uses the mltest library in the correct way, the ML Test Score would be 6. According to [1], this means that the system is "Reasonably tested, but it's possible that more of those tests and procedures may be automated." A developer can of course develop some more tests. These would be specific to the ML system at hand or the tests would be configured through e.g. a GitHub workflow rather than through unit tests, which are possible reasons why they are not provided in the *mltest* library. By developing only two more tests, the test score can be increased to 8, which would mean that the system has "strong levels of automated testing and monitoring, appropriate for

mission-critical systems" [1]. Therefore, the *mltest* library can be a useful tool to properly test any machine learning system.

6 Discussion

We were able to set up a library of general tests targeted at machine learning issues. For now, this library is nothing more than a directory with Python files full of useful methods that implement the described tests. For a developer to use this in another project, they would have to copy the directory. In the future, it would be useful to publish this library on its own, in such a way that it can easily be imported by developers and such that extensions can easily be distributed to the users.

The *mltest* library can be further expanded to cover more of the test concepts from [1]. Most of the tests that are not covered are dependent on access to the environment and infrastructure in which the models operate. If these tests could be generalized to the library there is a valuable opportunity. Furthermore, returning the “ML Test Score", as described in section 4, as an automatically calculated metric is a functionality that would make the library more user-friendly. This functionality is currently not implemented since we saw no general way to do this from the described directory of test functionalities. Perhaps, this directory with tests should become part of a much bigger “zoomed out" testing system that implements all tests from [1], including the tests that could not be implemented from the limited scope of the current testing directory.

7 Summary

Since a machine learning system is fundamentally different from testing a classical software system, testing the systems is also different. For example, data plays a much more prominent role in machine learning systems. To make testing your ML system easier, we proposed and developed the *mltest*. This library provides general test functionalities that can help with building a proper test suite for many ML systems. By making use of the *mltest* library and some extra efforts, developers can relatively easily make sure that their system has “strong levels of automated testing and monitoring, appropriate for mission-critical systems" [1].

Finally, we also provide an example machine learning project that integrates continuous testing with a full CI/CD pipeline and that serves a model that can predict tags for Stack-Overflow questions. This system was built from [2].

Acknowledgements

This project was developed for the Release Engineering for Machine Learning Applications course (CS4295) at Delft University of Technology. We would like to thank Luís Cruz for his feedback and guidance.

References

[1] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. 2016. What’s your ML test score? A rubric for ML production systems.
[2] Cruz Luis. 2022. Multilabel classification on Stack Overflow tags. https://github.com/luiscruz/remla-baseline-project.
[3] S. Karnani, J. Molhoek, Z. Nawar, A. van Steenweghen, and M. Zhang. 2022. Multilabel classification on Stack Overflow tags. https://github.com/ZiadNawar/remla-project.
[4] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2022), 1–36. https://doi.org/10.1109/tse.2019.2962027