DuckDQ: Data Quality Assertions for Machine Learning Pipelines

Till Döhmen [1] **Mark Raasveldt** [2] **Hannes Mühleisen** [2] **Sebastian Schelter** [3]

Abstract

Data quality validation plays an important role in ensuring the proper behaviour of productive machine learning (ML) applications and services. Observing a lack of existing solutions for quality control in medium-sized production systems, we developed DuckDQ: A lightweight and efficient Python library for data quality validation, that seamlessly integrates with scikit-learn ML pipelines and does not require a distributed computing environment or ML platform infrastructure, while outperforming existing solutions by a factor 3 to 40 in terms of runtime. We introduce the notion of data quality assertions, which can stop a pipeline when quality constraints of the input data or the model's output are not met. Furthermore, we employ stateful metric computations, which greatly enhance the possibilities for post hoc failure analysis and drift detection, even when the serving data is not around anymore.

1. Introduction

The development of machine learning (ML) models is a highly experiment-driven process. Data scientists typically leverage a combination of open source libaries like Pandas, scikit-learn, or tensorflow/pytorch for this, iteratively develop their models using computational notebooks like jupyter, and manage their data outside of relational database systems.

Data validation for machine learning. While this ecosystem provides data scientists with a high degree of flexibility, there is a lack of best practices such as version control, testing, and debugging. It is thus no surprise that many real-world ML projects struggle with the transition from development to production (Sculley et al., 2015). In contrast to classical software systems, the behaviour of ML-based systems is mainly determined by data, and errors in training or prediction data can lead to wrong and non-sensical behaviour if no safeguards are in place. To ensure the proper operation of ML-based applications, code-tests are not sufficient, the data itself must be checked for consistency with assumptions made in training and prediction pipelines (Polyzotis et al., 2017). Features and target values for ML models are typically extracted from heterogeneous data sources (e.g. log files or crawled data) and transformed into a single table via ETL/ELT processes, where data errors can easily be introduced and go undetected if no safeguards are in place.

Shortcomings of existing approaches. Existing solutions for validating ML data suffer from several disadvantages: (i) Solutions like Deequ (Schelter et al., 2018) are explicitly designed for very large datasets, and require cloud infrastructure as well as manual integration into training and serving systems. Additionally, the dependency on the distributed computing framework Apache Spark introduces large overheads for small- to medium-sized data; (ii) Enterprise approaches like TensorFlow Data Validation (TFDV (Caveness et al., 2020)) are tightly integrated with full-blown end-to-end ML platforms such as Google TFX, and are difficult to use "in the wild" outside of these platforms. (iii) Finally, Python-based approaches like great_expectations[1] or hooqu[2] are more lightweight, but not integrated with the ML development process and exhibit low performance for validations (Section 3). In runtime-critical production settings, this poses an unnecessary bottleneck for users wanting to safeguard ML applications against data errors.

The case for DuckDQ. As a consequence, we propose *DuckDQ*, a Python library to assist data scientists and data engineers with equipping ML models with data quality checks, without imposing the need for a machine learning platform infrastructure or distributed computing environments. DuckDQ provides a novel integration of data validation into the popular ML pipeline abstraction of scikit-learn. This allows data scientists to serialise "data assertions" together with a trained ML model, and execute them as part of the prediction pipeline, in order to safeguard the model from data errors. For the definition of data assertions, DuckDQ adopts an established fluent API for data quality validation,

[1]Fraunhofer FIT, Aachen, Germany [2]Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands [3]University of Amsterdam, Amsterdam, Netherlands. Correspondence to: Till Döhmen <till.doehmen@fit.fraunhofer.de>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

[1]https://greatexpectations.io
[2]https://github.com/mfcabrera/hooqu

known from Deequ's data unit tests (Schelter et al., 2019b). In contrast to existing solutions in the Python ecosystem, DuckDQ heavily optimizes the execution of its validation queries, which significantly reduces the runtime for data quality validation across different computational backends. It leverages the embeddable analytical database management system DuckDB (Raasveldt and Mühleisen, 2019) for efficient integration with the popular Pandas library and scikit-learn pipelines. Furthermore, DuckDQ computes and logs all validation metrics in a stateful manner. That allows computing metrics across multiple prediction batches without looking at the data twice or even needing the original data around. An approach, which can greatly enhances the opportunities for monitoring tasks (s. Section 2).

2. Overview of DuckDQ

In the following, we describe the design and implementation of DuckDQ and discuss its integration with the Python-based data science ecosystem.

Integration with scikit-learn pipelines. To facilitate the usage of DuckDQ in the Python ML ecosystem, we integrate DuckDQ with scikit-learn, one of the most popular ML libraries for Python. Scikit-learn popularised a pipeline abstraction, which allows users to chain up multiple data preprocessing steps and model training/prediction into a serialisable pipeline. Our DQPipeline module builds upon this paradigm and adds the twist that both, input and output of the pipeline can be automatically validated against user-defined data quality constraints. In case of a failed validation, the pipeline will raise a DataQualityException (optional) and prevent the pipeline from returning results. We call this mechanism "Data Assertions", in an analogy to assertions in software development.
DQPipelines can be used as a drop-in replacement for regular scikit-learn pipelines and all assertions will be automatically evaluated whenever the ML pipeline is trained and/or predictions are computed with the trained pipeline (s. Figure 1).

```
inp_assert = Assertion(CheckLevel.EXCEPTION)
  # check date format and positive trip distance
  .has_pattern("date_time",
              r"(\d{4}-\d{2}-\d{2}␣\d{2}:\d{2}:\d{2})")
  .is_positive("trip_distance")
  # check median trip distance via external function
  .has_quantile("trip_distance",
                0.5,
                lambda x: x <= expected_dist())
outp_assert = Assertion(CheckLevel.WARNING)
  # check ratio of positives is less than 10%
  .has_histogram_values("y",lambda x: x["1"].ratio < 0.1)

pipeline = DQPipeline([
  ('feature_encoding', Pipeline(...))
  ('clf', RandomForestClassifier())],
  input_assertion=inp_assert,
  output_assertion=outp_assert)

model = pipeline.fit(X_train, y_train)
```

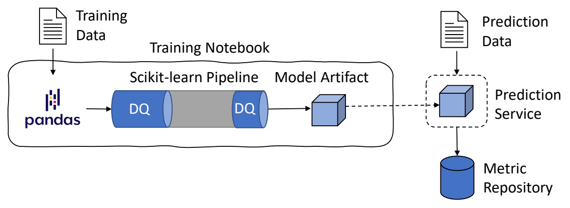


Figure 1. DuckDQ's (DQ) scikit-learn ML Pipeline integration.

The code snippet above shows an exemplary pipeline, which triggers an exception, when the `date_time` attribute does not match the given pattern, the `trip_distance` attribute has non-positive values or when its 50th percentile is greater than an expected distance (which is calculated via an external function). Furthermore, when more than 10% of the model's predictions are positive, it will generate a warning message.
A model equipped with a data assertion can be serialized using the Python library *dill* and shipped to other systems, e.g., from a development to a production environment. Users can thus equip their trained ML pipelines with integrated data safeguards when deploying them, which raise an exception or warning if data quality conditions are not met. Note that the DuckDQ library and its Python dependencies need to be installed on the serving system as well.
The following snippet shows an example of a user loading the trained pipeline in a production environment and making a prediction on live data, while triggering the output assertion. The ratio of positive predictions is 14%, but it should have been $<10\%$.

```
model = dill.load(model_artifact_path)
predictions = model.predict(live_data)

Warning: Data quality validation failed. Details:
Output Check: WARNING
    ComplianceConstraint(Histogram("y")): FAILURE (0.14)
```

Stateful computations for debugging and drift detection. With each model training and prediction, we record metadata of the validation into the metric repository - a local database. Besides validation results and individual metrics, we record intermediate states of the metrics computation, which entails that all metrics can be reconstructed and aggregated across multiple training/prediction runs, without requiring access to the raw data. This is particularly useful for metrics like mean, standard deviation, and relative frequency histograms, which, when fully computed can conventionally not be merged across multiple data batches of different sizes. In a resource-constrained environment, like an IoT/edge device, it might not be desirable to hold raw prediction data for a long period of time, on the other hand, users might still want to retrieve aggregated metrics from e.g. the previous day or week for monitoring and debugging purposes. As intermediate states are typically much smaller than the raw data it is feasible to store them locally for a longer period of time. This also opens up opportunities for integrated drift detection, as summary statistics can be tracked and compared easily across long periods of time, covering hundreds of thousands of prediction runs, without the requirement to store raw data locally or to process raw data more than once. In summary, we deem stateful computation of metrics in combination with a local repository, as implemented in DuckDQ, to be a particularly useful approach for failure analysis and monitoring tasks. Stateful metric computations are described in more detail in (Schelter et al., 2019b).

DuckDB for efficient querying of DataFrames. For the evaluation of data assertions, we leverage DuckDB. The Python data science ecosystem is primarily centered around Pandas DataFrames, which makes DuckDB (Raasveldt and Mühleisen, 2019) a perfect fit for several reasons: (i) DuckDB is an SQL engine, which allows us to apply battle-tested multi-query optimization techniques from Deequ (Schelter et al., 2019b); (ii) DuckDB has been explicitly designed for tight integration with the Python Data Science ecosystem. In particular, it allows us to register a Pandas DataFrame as a virtual table, and to query it in a zero-copy manner without significant performance overhead compared to physical tables; (iii) Furthermore, DuckDB is an embeddable database engine, which does not have any external dependencies and can run seamlessly within a Python process; (iv) DuckDB uses an efficient columnar-vectorized query execution model which yields great performance advantages for analytical queries, compared to native Pandas DataFrame operations and row-oriented relational database backends.

Architecture. DuckDQ's core library consists of (i) a verification logic component, and (ii) a computational engine component, which can be equipped with different execution engines (Figure 2). We implemented an SQL-based DuckDB engine and factored out generic parts to a database-agnostic SQL-engine. As a side effect, this engine can be utilized for general-purpose data quality validation on other SQL-based RDBMS[3].
DuckDQ's engines map data quality constraints from the verification logic to internal *operators*, which define how the queries to the database are built. Inspired by Deequ (Schelter et al., 2019b), there are two different kinds of operators: 1.) *Scan-sharing-operators* and 2.) *Group scan-sharing-operators*. Scan-Sharing operators are the primarily used type, covering about 90% of all constraints. They define a set of aggregations (e.g., the mean-operator leverages sum and count), optionally combined with a row-filter (`CASE WHEN...`). All scan-sharing operators

[3]Currently, all databases supported by the SQLAlchemy (`https://www.sqlalchemy.org`) connector.

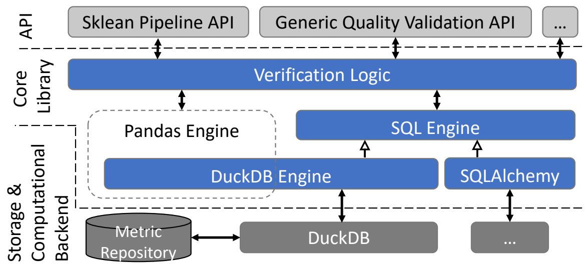


Figure 2. System architecture of DuckDQ.

can be executed within a single table scan. Hence, expensive repetitive table scans are avoided. Group-scan-sharing operators define aggregations over grouped data and allow for the sharing of scans for all operators that require the same grouping. An example of such an operation is the calculation of the uniqueness of a column.

3. Evaluation

In order to validate our design decisions regarding runtime efficiency, we compare DuckDQ against three other Python-based data quality validation frameworks[4] mentioned in Section 1: PyDeequ, hooqu and great_expectations (GrExp.). It is to note that all experiments were conducted on a conventional desktop machine with an Intel i7 8-Core CPU and 32GB DDR4 RAM.

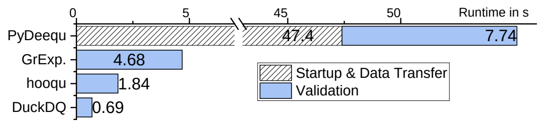
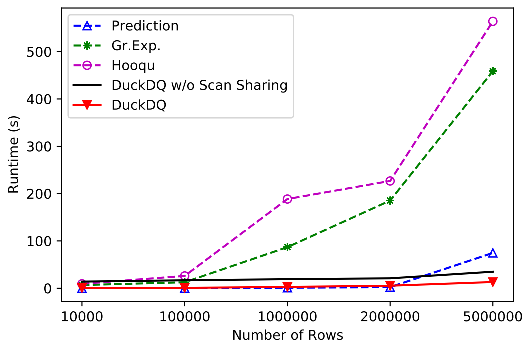


Figure 3. Runtimes for data validation on a Pandas DataFrame, with approx. 3M rows and 8 realistic quality constraints. DuckDQ outperforms compared solutions by factor 3.

Real World Data. First, we use a medium-sized real-world dataset with approximately 3M rows and ten columns from a hotel recommendation use case[5]. We read data from the CSV file into a Pandas DataFrame, as it would be typically done. Then we pass it to the respective systems and start a data quality validation with five completeness constraints on four numeric columns and one string column, as well as five uniqueness constraints on each of the remaining columns (s. Figure 3). PyDeequ shows a significant overhead due to the initialization of Spark and the data transfer from Pandas to Spark DataFrames. Also, the subsequent

[4]Since TDFV follows a different API and design than the other tools, we could not find a fair way to compare it to the others.

[5]`https://www.kaggle.com/c/expedia-hotel-recommendations/data`



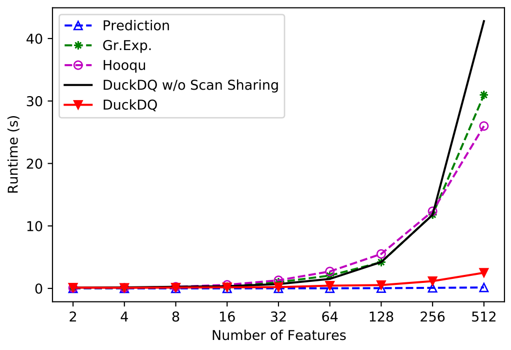


Figure 4. Runtime of data quality validation by number of rows (left), and runtime of data quality validation by number of features (right).

validation is slower compared to the other systems, which confirms that the scalability of Deequ comes at large costs on small- to medium-sized data. The libraries hooqu and great_expectations, which both use Pandas to compute the required data summary statistics, show runtimes between 1.84s and 4.68s, respectively. DuckDQ shows a roughly 3-fold lower execution time at 0.69s.

Varying Number of Rows/Attributes. In a second experiment, we investigate the runtime properties of DuckDQ relative to the size of the dataset and the number of attributes to be checked. We use a built-in function of scikit-learn to generate different regression datasets with $[0.1M - 5M]$ rows and $[2 - 512]$ attributes and evaluate three simple constraints on each attribute with each of the libraries. PyDeequ was excluded from the comparison due to excessive runtime - instead, a small ablation study was included with a version of DuckDQ that does not use the scan sharing optimization. For comparison, the runtime of a linear regression batch prediction for the same dataset was included. The left side of Figure 4 illustrates the runtime over different dataset sizes while keeping the number of validated attributes fixed at 512. It is observed that the runtime of hooqu and great_expectations grows steeply with the number of rows, far exceeding the prediction runtime, while the validation runtimes of both versions of DuckDQ stay in the same order of magnitude as the bare prediction runtime. This can be attributed to the fact that great_expectations and hooqu rely on Pandas to determined the required summary statistics, while both DuckDQ versions leverage DuckDB to compute the same.
When increasing the number of attributes, and keeping the dataset size fixed at 0.1M (right side of Figure 4), one can observe the effect of the scan sharing optimization. The runtime of hooqu, great_expectations as well as DuckDQ w/o Scan Sharing growth linearly with the number of attributes, while the DuckDQ execution time, again, stays in the same order of magnitude as the prediction runtime. That is because DuckDQ requires only one full data scan to determine all metrics, while great_expectations and others require $n = number\_of\_attributes$ scans.

In summary, we find that by using well-chosen optimization techniques and by leveraging the analytical database engine DuckDB, we can increase the efficiency of Pandas-based data quality validation between 3- and 40-fold compared to existing solutions, without requiring distributed computing environments or any additional data roundtrips.
The source code of DuckDQ and the experiments are publicly available [6] [7].

4. Conclusion

We presented *DuckDQ*, a lightweight Python library to safeguard production ML models against data errors, e.g. when running ML prediction pipelines in a Python-based web server backend. We showed that existing solutions introduce a major bottleneck in such pipelines for datasets >100k rows and >64 attributes, while DuckDQ's validation runtime stays in the same order of magnitude as the prediction runtime. DuckDQ is particularly well suited for medium-sized use cases in resource-constrained environments, where distributed computing is not an option and/or ML platform infrastructure is not available. It features seamless integration with the Pandas data science ecosystem and offers a range of opportunities for debugging and monitoring, exploiting the stateful computation of metrics, which allows computing summary statistics incrementally across multiple prediction-runs, without needing to keep raw data around.
We hope that this work can contribute to a more widespread adoption of MLOps best practices, in particular for users which have to work in resource-constrained environments and/or with low operational costs, like citizen data scientists or small companies/teams venturing into production ML.

[6] https://github.com/tdoehmen/duckdq
[7] https://github.com/tdoehmen/duckdq-exp

Acknowledgements

This work has been partially funded by the European Union Horizon 2020 Research and Innovation program through the DEMETER project (Grant Agreement No. 857202).

References

Emily Caveness, Paul Suganthan G. C., Zhuo Peng, Neoklis Polyzotis, Sudip Roy, and Martin Zinkevich. 2020. TensorFlow Data Validation: Data Analysis and Validation in Continuous ML Pipelines. *SIGMOD*, 2793–2796.

Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data management challenges in production machine learning. *SIGMOD*

Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. *SIGMOD*, 1981–1984.

Sebastian Schelter, Dustin Lange, Meltem Celikel and Philipp Schmidt. 2018b. Automating Large-Scale Data Quality Verification. *PVLDB* 11 (12).

Sebastian Schelter, Stefan Grafberger, and Dustin Lange. 2019b. Differential Data Quality Verification on Partitioned Data. *ICDE*, 1940–1945.

David Sculley et al. 2015. Hidden technical debt in machine learning systems. *NeurIPS*, 2503–2511.