CONTENTS

ABSTRACT 1
1. INTRODUCTION 1
2. FRAMEWORK DESIGN 2
2.1. EXAMPLE TASKS 2
2.2. DATA CORRUPTIONS 2
2.3. EVALUATORS 2
3. USAGE AND CUSTOMISATION 3
3.1. EVALUATING THE IMPACT OF DATA ERRORS 3
3.2. CUSTOM TASKS AND DATA CORRUPTIONS 3
4. EXAMPLE USE CASES 3
4.1. MEASURING THE ROBUSTNESS OF A MODEL 3
4.2. STRESSTESTING INTEGRITY CONSTRAINTS 4
5. RELATED WORK 6
6. LEARNINGS & CONCLUSION 6
REFERENCES 6

JENGA - A Framework to Study the Impact of Data Errors on the Predictions of Machine Learning Models

Sebastian Schelter*
University of Amsterdam
s.schelter@uva.nl

Tammo Rukat
Amazon Research
tammruka@amazon.com

Felix Biessmann†
Einstein Center Digital Future
fbiessmann@beuth-hochschule.de

ABSTRACT

Machine learning (ML) is increasingly used to automate decision making in various domains. Almost all common ML models are susceptible to data errors in the serving data (for which the model makes predictions). Such errors frequently occur in practice, caused for example by program bugs in data preprocessing code or non-anticipated schema changes in external data sources. These errors can have devastating effects on the prediction quality of ML models, and are, at the same time, hard to anticipate and capture.

In order to empower data scientists to study the impact as well as mitigation techniques for data errors in ML models, we propose Jenga, a light-weight, open source, experimentation library. Jenga allows its users to easily test their models for robustness against common data errors. Jenga contains an abstraction for prediction tasks based on a dataset and a model, an easily extendable set of synthethic data corruptions (e.g., for missing values, outliers, typos and noisy measurements) as well as evaluation functionality to experiment with different data corruptions.

Jenga supports researchers and practitioners in the difficult task of data validation for ML applications. As a showcase for this, we discuss two use cases of Jenga: studying the robustness of a model against incomplete data, as well as automatically stress testing integrity constraints for ML data expressed with tensorflow data validation.

1 INTRODUCTION

Many companies and organisations are moving to a data-driven approach, where machine learning (ML) is used to assist and automate decision making in various domains. Yet the application of ML in production settings often faces a number of pitfalls. Almost all common ML models are susceptible to data errors in the serving data (for which the model makes predictions). Such errors frequently occur in practice, caused for example by program bugs in data preprocessing code or non-anticipated schema changes in external data sources. These errors can have devastating effects on the prediction quality of ML models [12], and are, at the same time, hard to anticipate and capture. While many aspects of the impact of data changes on ML models are studied in the ML literature [1, 7, 9], it can be difficult to relate this research to the errors occurring in practical ML applications, as these approaches all require distributional assumptions about the change.

Data errors in production machine learning. Frequently, the errors in production deployments do not originate from changes in the data generating real-world processes, but from programming errors in the data pipelines constructing the serving data [4] or from errors during data integration from different sources [6, 21]. Such errors often only become apparent once models are deployed in complex production use cases [16].

We have come across several real world instances of such data errors. In one case, a linear model had been trained on demographic data (including a person's age), and the age value had been missing for some records for which the model should supply predictions. A software engineer (without knowledge of the model intricacies) then wrote preprocessing code to replace all missing age values with zeroes, the default value for initialising integers in many programming languages. This led to a unwanted misbehavior of the model, which effectively treated all these records as "toddlers". In another case, we learned about an ML model where the pipelines for training and serving data were running in different cloud environments. As a result, the code bases for data preparation on the training and serving side accidentally diverged, which introduced hard to detect data errors. Such errors can have devastating impact, as all guarantees about the reliability of the predictions of the ML model may be lost, which can lead to monetary losses (e.g., if buying decisions are made based on the predictions of a forecasting model) and bad user experiences (e.g., if users are presented with non-sensical recommendations in an online shop).

Evaluating the robustness of models against common data errors. These examples show the need for testing the robustness of ML models to data errors before they are deployed to production. Recent research focuses on detecting and handling such data errors, e.g., by proposing unit tests and integrity constraints for ML data [4, 17], ML-based missing value imputation [2] and validating the predictions of black box models [19]. In our experience, it is difficult to provide broadly valid empirical evaluations of these approaches, and to generate synthetically corrupted data that represents the scenarios that we encounter in the real world.

To address this need, we design the *Jenga* library, which we present in this paper. Jenga enables data scientists to study the robustness of their models against errors commonly observed in production scenarios. Based on the findings from experimenting with Jenga, users can take appropriate measures to protect their deployed models against impactful data errors, e.g., with custom integrity constraints implemented via tensorflow data validation [4].

In summary, this paper provides the following contributions.

- We introduce our open source framework Jenga to study the impact of data errors on ML models (Section 2).
- We describe how to implement custom prediction tasks and synthetic data corruptions in Jenga (Section 3).
- We discuss two use cases for Jenga: studying the robustness of a model against incomplete data, and automatically stress testing integrity constraints for ML data expressed with tensorflow data validation (Section 4).

*work done while at New York University
†work done while at Amazon Research and Beuth University, Berlin, Germany

Jenga is publicly available under an open source license at https://github.com/schelterlabs/jenga.

2 FRAMEWORK DESIGN

We introduce the design of Jenga. The goal of Jenga is to enable data scientists to evaluate the impact of data errors on their models, and to evaluate techniques that make these models more robust. We design Jenga around three core abstractions: (*i*) *tasks* contain a raw dataset, an ML model, and represent a prediction task; (*ii*) *data corruptions* take raw input data and randomly apply certain data errors to them (e.g., missing values); (*iii*) *evaluators* take a task and data corruptions, and execute the evaluation by repeatedly corrupting the test data of the task, and recording the predictive performance of the model on the corrupted test data.

We provide three sample tasks (Section 2.1), several data corruptions (Section 2.2) and two different evaluators (Section 2.3) as part of the framework.

2.1 Example Tasks

We provide three exemplary prediction tasks in Jenga. Note that users can define and implement their own custom tasks with low effort (see Section 3 for details). We choose simple binary classification tasks for product review classification (predicting whether the review of a video game was deemed helpful), income estimation (predicting whether a person earns more than $50,000 per year based on demographic data) and image recognition (distinguishing sneakers from ankle boots), which resemble real world use cases, and leverage publicly available datasets and widely used ML models. We focus on relatively small-scale problems, which do not require costly infrastructure (e.g., the models can be trained in a couple of minutes on a multicore CPU), in order to allow users to rapidly experiment and play with our framework. These tasks are meant as examples to enable users to test our data corruptions and evaluators, and serve as a template for our users to integrate Jenga with their own custom prediction tasks.

2.2 Data Corruptions

In the following, we describe the types of data corruptions available in Jenga. Each error type requires the specification of a column c to be affected by the error and a fraction of rows $r \in [0, 1]$ that should be affected.

Corruption sampling.: Whether or not a value is affected by a corruption is often the result of errors in complex preprocessing pipelines. In order to account for realistic corruption patterns we model the fraction of rows affected by a corruption as follows. A value x_c in column c is corrupted (*i*) *independent of other values* (corrupted values are sampled *completely at random*), (*ii*) dependent on values in columns other than c (corrupted values are sampled *at random*), or (*iii*) dependent on values in column c (corrupted values are sampled *not at random*). This modelling is inspired by literature on missing value imputation, where three types of missingness are commonly distinguished [10]. As these three sampling procedures a can capture the complex error patterns often observed in practice we chose to make it applicable not only to missing value corruptions, but to all other error types as well.

Missing values. Missing values are amongst the most common data errors in practice. Missing values can have devastating effects on training and prediction, depending on how a data pipeline deals with missing values before feeding the data to a downstream ML model. An important factor for the impact of missing values are the missingness patterns, described in the previous paragraph, *missing completely at random (MCAR)*, *missing at random (MAR)* and *missing not at random (MNAR)*.

We additionally support the injection of *missing values based on "prediction difficulty"*, where we consider the fact that there ML model downstream that is affected by missing data. This error type considers the entropy of the ML model predictions for the data rows and discards values based on their difficulty for the model, akin to uncertainty sampling in active learning.

Swapped values. We replace a specified ratio of values in one column with values in another column. This corruption mimics users mixing up entries in input forms [6] or programming errors in data preparation code, where a programmer accidentally swaps target columns to write to.

Scaling. We randomly scale a subset of the values by 10, 100 or 1000. This perturbation mimics cases where the scale of an attribute is accidentally changed in preprocessing code (e.g., because a developer accidentally changes the code to record durations in milliseconds instead of seconds).

Noise. We corrupt a fraction of a column's values by adding gaussian noise centered at the data point with a standard deviation randomly selected from the interval of 2 to 5. This corruption is intended to mimic measurement errors.

Encoding errors. This corruption replaces certain characters in string attributes (e.g., a with á), and is meant to simulate encoding errors, e.g., for data retrieved from web pages which indicate a false encoding.

Image corruptions. Dealing with corrupted training images is a well studied problem in computer vision [3] for which a lot of tooling exists already. We therefore integrate existing image corruptions from the *augmentor*[1] library into jenga.

2.3 Evaluators

Finally, Jenga provides so-called *evaluators*, which measure the impact of data corruptions on the model's predictive performance. Jenga currently features two evaluators: The `CorruptionImpactEvaluator` takes a provided task, a trained model and a manually specified list of corruptions. It applies each data corruption to the held-out test set of the task and computes the predictive performance of the model in light of the data corruption. We show how to use this evaluator with a few lines of code in Section 3.1, and discuss a detailed example of measuring the impact of missing values on a task in Section 4.1.

The second evaluator allows users to additionally integrate a data validation schema into the evaluation. In many cases, it is not possible to make an ML model completely robust against data errors [19]. A common approach to prevent feeding corrupted data to deployed ML models is to run data validation checks on the serving data on which the model is applied. Popular libraries for this task are *tensorflow data validation (TFDV)*[2] [4] or Deequ[3] [18]. They allow users to define a schema and constraints for the serving data (e.g., that a given attribute must not contain missing values), and efficiently execute this check before passing the data to an ML model. Jenga contains a custom `SchemaStresstestEvaluator` for feature data validation with a TFDV schema. This evaluator works analogous to the previous

[1] https://github.com/mdbloice/Augmentor
[2] https://www.tensorflow.org/tfx/guide/tfdv
[3] https://github.com/awslabs/deequ

one, but additionally records whether the check of a provided data validation schema would have correctly detected the negative impact of the data corruption. We provide an extensive example for this evaluator in Section 4.2.

3 USAGE AND CUSTOMISATION

We implement Jenga based on several popular open source ML frameworks in python. We leverage *pandas* for data wrangling, and *numpy* for numerical computations. We implement feature extraction and preprocessing via *scikit-learn*'s pipeline abstraction, and also use classical ML models from this library. We rely on *keras* and *tensorflow* for defining and training neural networks.

In the following we first give an example of how to use Jenga to evaluate the impact of data corruptions (Section 3.1), and subsequently discuss how to implement custom tasks and data corruptions in Jenga's API in Section 3.2.

3.1 Evaluating the Impact of Data Errors

The core use case of jenga is to evaluate the impact of certain data corruptions on a prediction task. This can be implemented with a few lines of code: We have to instantiate the task and data corruptions that we want to evaluate, and can execute the evaluation with Jenga's `CorruptionImpactEvaluator`. This allows us to measure the impact of a predefined list of data corruptions on the predictive performance of a model.

```
# Create the prediction task
task = IncomeEstimationTask()
# Train a baseline model
model = task.fit_model(task.train_data,
 task.train_labels)
# Specify the data corruption to test
corruption = MissingValues(column='age',
 missingness='mcar', fraction=0.05)
# Create the evaluator
evaluator = CorruptionImpactEvaluator(task)
# Run the evaluation with 10 repetitions
result = evaluator.evaluate(model,
 num_repetitions=10, corruption)
# Impact on predictive performance
print(f""" Score on
  clean data:     {result.baseline_score}
  corrupted data: {result.corrupted_scores} """)
```

Here, we setup a task, train the corresponding model and define the corruption that we are interested in. We provide these to the evaluator together with the specification of the number of repetitions to execute for each corruption. The evaluator repeatedly corrupts the (copied) test data of the task, computes the prediction quality of the model on the corrupted data and finally provides a result object with the corresponding scores for each corruption to investigate. Note that we could also have specified more than one data corruption to evaluate.

3.2 Custom Tasks and Data Corruptions

We design Jenga with the goal to make it easy for data scientists to wrap their existing code as a prediction task, which allows them to reuse our data corruptions and evaluators. In addition, we also make it easy to design custom data corruptions. Therefore, we next describe how to implement the two basic building blocks of Jenga, a `Task` and a `DataCorruption`.

Implementing a custom task. Jenga allows data scientists to implement custom tasks with low effort. We provide an abstract base class `ClassificationTask` with two methods that users must implement. In the constructor, users have to load the input data for the task. Next, they have to implement the `fit_model` method, which trains the accompanying prediction model for the task from training data provided in a pandas dataframe. The model produced by this must support scikit-learns predictor API. Finally, the `score_on_test_data` must be implemented, which computes the desired metric for the task (e.g., ROC AUC) from the predicted label probabilities of the model.

Implementing a custom data corruption. At the core of Jenga are data corruptions, whose impact on the predictive performance of a model we want to investigate. Data corruptions transform a dataframe into another dataframe with potentially corrupted values. We provide an abstract base class, `DataCorruption`, that users can extend by providing only a single method, called `transform`. In the following listing, we implement a data corruption that mimics a case where duration that needs be expressed in seconds is accidentally recorded in milliseconds (e.g., scaled by a factor of 1000) in a fraction for the rows.

```
class MillisInsteadOfSeconds(DataCorruption):
 ...
 def transform(self, data):
  # Operate on a copy of the data
  corrupted_data = data.copy(deep=True)
  # Pick a random fraction of the rows
  rows = np.random.uniform(len(data)) < self.fraction
  # Multiply the column values of the chosen rows
  corrupted_data.loc[rows, self.column] *= 1000
  return corrupted_data
```

We first conduct a deep copy of the input data, which we will corrupt later on. Then, we randomly pick the indexes of the rows that we want to corrupt, and finally mulitply their values by a 1000 to mimic milliseconds. Note that tasks and data corruptions implemented with our API can be readily used in the existing evaluators from Jenga, as outlined in Section 3.1.

4 EXAMPLE USE CASES

We discuss two examplary use cases of our framework that resemble real world problems which we encountered in production ML applications. Note that we provide implementations (in the form of Jupyter notebooks) for all these use cases in our github repository at https://github.com/schelterlabs/jenga.

4.1 Measuring the Robustness of a Model against Incomplete Data

Overview. In our first experiment, we showcase how to study the impact of missing values on the predictions of a model. This targets a common usage scenario, where data scientists, who have a trained model in production (or ready for production), want to study its robustness towards incomplete data with Jenga. They want to reach a conclusion on how well the model itself (in combination with different missing value imputation methods) can mitigate the impact of missing values in the serving data. Incomplete data is a very common issue in real world deployments, where data is often missing as a result of programming errors, data integration issues or unanticipated schema changes in an external data source.

Setup. We experiment with a logistic regression model for our income estimation task from Section 2.1. The goal of this task is to predict from demographic data whether an individual has a high income. We train a model on clean training data, and evaluate its predictive performance (in terms of ROC AUC) on test data with synthetically injected missing values. We focus on four categorical attributes in the data: `education`, `marital_status`,

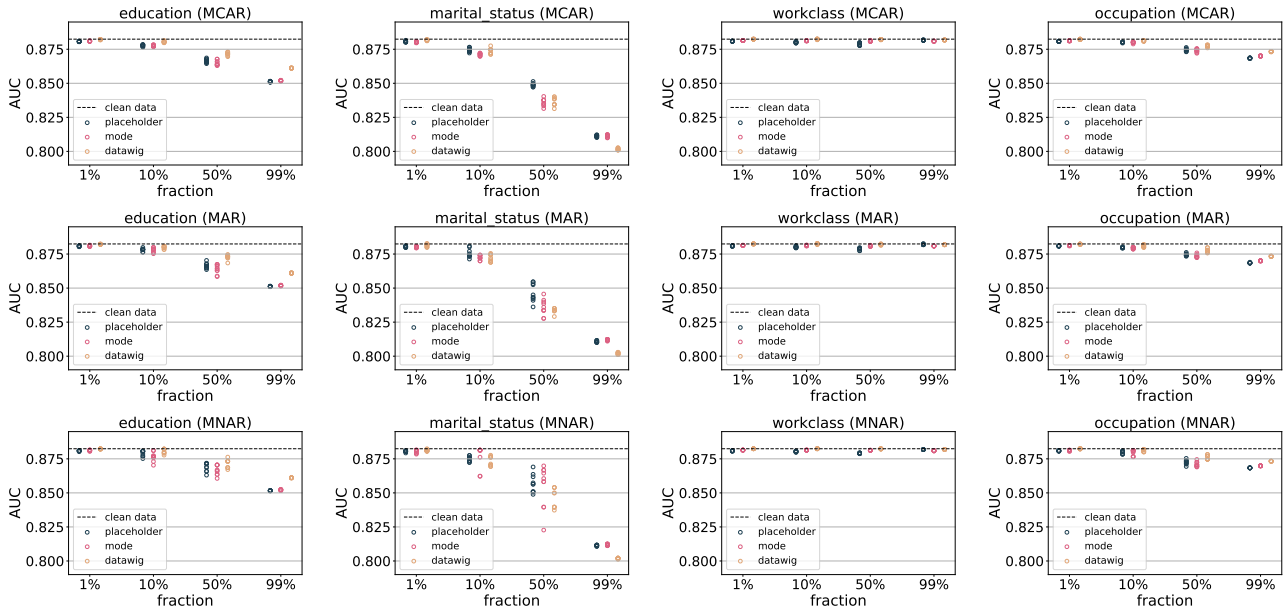

Figure 1: Evaluation of the robustness of a model for the income estimation task against incomplete data. We plot the AUC score achieved with different missing value imputation strategies (*placeholder*, *mode* and *datawig*) against the fraction of injected missing values. The impact differs by attribute and there is no clear dominating imputation strategy, indicating that it is difficult to make the model fully robust against this type of data error.

`workclass`, and `occupation`, and inject missing values into 1%, 10%, 50% and 99% of randomly chosen values of a given attribute.

We repeat this process for all three kinds of missing values ("missing completely at random" (MCAR), "missing at random" (MAR), "missing not at random" (MNAR)) as discussed in Section 2.2. We repeat each individual configuration ten times, and report the performance on corrupted test data (in comparison to the performance on clean data), where we differentiate between three different ways to make the model handle the missing values:

- First, we replace missing values with a constant *placeholder* symbol.
- Secondly, we replace missing values with the *mode* (the most frequent value in the column) via scikit-learn's `SimpleImputer`
- Thirdly, we train a dedicated ML model to impute missing values based on the structure present in the complete records [2]. We leverage the *datawig* library[4], which automatically featurises tabular data and trains a neural network to predict the missing values.

Results. The experimental results are shown in Figure 1. We find that the impact of the missing values is highly dependent on the attribute we target. There is nearly no impact for `workclass`, a very minor impact for `occupation` for less than 50% missing values, a much stronger impact for `education`, and we encounter the highest impact for missing values in `marital_status`. We additionally see that the impact is in some cases different for different types of missing values, e.g., values "missing not at random" in the `marital_status` attribute seem to be easier to handle than the other types of missingness.

In summary, we find no clear dominating strategy for handling the missing values in this particular task. Having the model deal with the missing values via a placeholder symbol is simple and works well in many cases. However, there are some setups where leveraging a dedicated missing value imputation strategy helps, e.g., datawig for a high fraction of missing values in `education` or for `occupation`. We conclude that the model itself cannot handle missing values reliably in all cases, even in combination with imputation. Thus, the data scientists need to put checks in place to safeguard the serving data on which the model is applied.

4.2 Stresstesting Integrity Constraints for ML data

Overview. Our next experiment shows how to put safeguards in place for an ML model. This experiment applies a schema and constraints for ML data, and executes a stresstest for them, as discussed in Section 2.3. We leverage the product review classification task discussed in Section 2.1, where the goal is to predict whether uses found the review of a videogame helpful or not.

We train a model for this task, and additionally create a schema with integrity constraints for the test data in TFDV. Next, we run Jenga's `SchemaStresstest` which generates random data corruptions for the test data, and determines whether our schema catches these errors, and what the impact of these on the prediction quality of the model (in terms of ROC AUC) would have been.

In real world use cases, it is difficult for data scientists to come up with an appropriate schema and constraints for their data, and we develop our stresstest to uncover errors which are not caught by the current schema. As a consequence, data scientists can iteratively improve their integrity constraints until they pass the stresstest.

[4]https://github.com/awslabs/datawig

```
from jenga.tasks.reviews import VideogameReviewsTask
from jenga.evaluation.schema import SchemaStresstest
import tensorflow_data_validation as tfdv
# Setup task
task = VideogameReviewsTask()
# Create a schema to test
train_data_stats =
 tfdv.generate_statistics_from_df(task.train_data)
# Auto-infer schema from training data
schema = tfdv.infer_schema(statistics=train_data_stats)
# Manually adjust schema
review_date_feature =
 tfdv.get_feature(schema, 'review_date')
review_date_feature.distribution_constraints
 .min_domain_mass = 0.0
# Define model to include in stress test
model = task.fit(task.train_data, task.train_labels)
# Run stress test with 250 randomly generated
# data corruptions
stress_test = SchemaStresstest()
results = stress_test.run(task, model, schema,
 num_corruptions=250, performance_threshold=.03)
```

Setup. The code above shows the setup of the experiment. We generate a schema for the feature data of the task, with a semi-automatic approach, where we first have TFDV automatically infer a schema for the data (via `tfdv.infer_schema`).

The schema correctly identifies the data types and categorical domains of most of the attributes of the data. It is too strict however, as it does not account for the fact that all the values in the `review_date` column will change for future data. We manually adjust the schema for this attribute by setting the minimum domain mass that must be shared between the values found at schema inference time and the future values to zero, allowing new values to appear in the column. The following listing shows an excerpt of the schema and constraints for the data, containing type information, completeness requirements and domain values for the data attributes.

```
...
feature {
 name: "star_rating"
 type: INT
 presence { min_fraction: 1.0 } }

feature {
 name: "verified_purchase"
 type: BYTES
 domain: "verified_purchase"
 presence { min_fraction: 1.0 } }

feature {
 name: "review_date"
 type: BYTES
 domain: "review_date"
 presence { min_fraction: 1.0 }
 distribution_constraints { min_domain_mass: 0.0 } }
...
string_domain {
 name: "verified_purchase"
 value: "N"
 value: "Y"
}
...
```

We evaluate the schema with a stress test which applies 250 randomly generated data corruptions to the serving data of the model and measures their impact on the prediction quality.

Results. The model achieves an AUC of 0.78828 on clean data, and we consider all predictions on corrupted data with more than 3% decrease in prediction performance as failures. Jenga categorizes the results as following:

- True positives, where TFDV reports a schema violation and the prediction quality on the corrupt test data drops below the threshold.
- True negatives, where TFDV reports no schema violation and the prediction quality on the corrupt test data is within the threshold.
- False positives, where TFDV reports a schema violation, but the prediction quality on the corrupt test data does not drop below the threshold. Note that it might still make sense to capture and investigate these data errors, as they can be indicators of problems in preprocessing code or external data sources.
- False negatives, where TFDV reports does not report a schema violation, but the prediction quality on the corrupt test data does drop below the threshold. These are the most important findings from a stress test as they indicate data errors to which the model would be vulnerable in production. It is imperative to adjust the schema to catch these errors.

In the following, we list several findings from our stress test example in Table 1 and discuss them.

True positives. Out of the 250 corruptions, we find 88 true positives. For example, we find that the model crashes for missing values in the numeric `star_rating` column, and that the prediction quality drops more than 3% for gaussian noise in this column and for a large number of swapped values between the `verified_purchase` and `title` column. Note that all of these errors are correctly detected by TFDV.

error type	*column(s)*	*frac*	*comment*
True positives			
missing values	`star_rating`	.25	crash
swapped values	`review_body, vine`	.75	unseen values
swapped values	`verified_purchase, title`	.45	unseen values
missing values	`vine`	.53	incompleteness
gaussian noise	`star_rating`	.25	type (int to float)
True negatives			
encoding	`vine`	.72	no changes
encoding	`review_id`	.83	column not used
swapped values	`review_id, product_parent`	.17	columns not used
missing values	`product_id`	.27	column not used
False positives			
missing values	`vine`	.93	unseen values
gaussian noise	`star_rating`	.16	type (int to float)
swapped values	`product_id, marketplace`	.35	unseen values
encoding	`marketplace`	.72	unseen values
False negatives			
scaling	`star_rating`	.92	range check missing
encoding	`title_and_review`	.76	no encoding checks
missing values	`title_and_review`	.80	no length checks
swapped values	`title_and_review, review_headline`	.75	no length checks

Table 1: Results found by the schema stress test for detecting impactful data errors on the product review task.

True negatives. We additionally find 75 true negatives, which mostly include cases where a textual column is being corrupted which is ignored by TFDV, but also not used by the model, whose prediction quality is therefore not affected by the corruption.

False positives. We encounter 39 false positives. We for example see that even a high number of missing values in the `vine` column

do not strongly affect the prediction quality, as well as a low number of noisy values in the `star_rating` column or encoding errors in the `marketplace` column, which is not used by the model.

False negatives. The most important results from the stress test are false negatives, e.g., data corruptions that are not detected by our TFDV schema, but strongly affect the prediction quality of the model. In a real world use case, we need to extend our schema to catch all these errors. We see that scaling values in the `star_rating` column strongly affects the prediction quality. This is an indicator that we should add a range check for this column to our schema. Furthermore, all kinds of errors in the `title_and_review` column negatively affect the prediction quality. This is a textual column for which TFDV does not generate constraints automatically. Checks for both the length and encoding of the values in that column are required to capture the outlined errors.

We argue that it should become a best practice to execute such stresstests for data errors before putting ML models into production, and we think that such testing capabilities should be integrated into common ML deployment pipelines.

5 RELATED WORK

Addressing the challenges in productionizing ML models is a field with growing interest in recent years [2, 8, 12, 16, 20]. Several solutions were proposed for validating ML models and their predictions. Most of these originate from a statistical ML or a data management perspective. Approaches from the ML community are based on distributional assumptions about the data shift, such as label shift [13], and covariate shift [1]. These assumptions often seem inapt to describe practically relevant data changes for engineers, such as the errors described above. Moreover, the proposed methods often limit themselves to adapting a particular model or learning paradigm.

There exist several approaches from the data management community to validate the input data of ML pipelines. For example, Google's TFX platform [4] offers validation for input data via a feature schema, and Deequ [17] enables unit tests for data, but both of them do not quantify the potential impact of errors on the model predictions. On a related note, there is a growing body of work on model monitoring [19], model diagnosis [5] and model unit testing for neural networks [11].

6 LEARNINGS & CONCLUSION

During our work on real world ML deployments, we have repeatedly come across scenarios where data errors heavily impacted deployed models and applications.

Missing values in data can in some cases, propagate through various connected pipelines until a customer facing model crashes, and it is very tedious to trace these errors back to the original data source which introduced the missing values. In internationalised applications, which operate on text in non-western languages, it is common to encounter encoding issues which are often introduced by a wrongly configured intermediate data store, and are again very hard to pinpoint and fix. Another common source of errors is calendar-related data, where dates and durations are often incorrect, e.g., due to movable holidays. Furthermore, often ML models are trained by specialised teams, and then handed over to business teams. In such cases, we often experienced that the data provided to the ML experts had not been sampled in a representative way by the business team, and as a consequence, the resulting model will not perform well later on due to the dataset shift introduced by the non-representative sampling.

These experiences motivate our presented library Jenga, which enables data scientists to evaluate the performance of ML models under data errors. Jenga builds on existing ML libraries, and allows practitioners and researchers to quickly build ML testing suites for their models with a broad range of data errors that we observed over several years of maintaining ML applications. We think that is is necessary to establish a set of best practices for testing ML models, analogous to established best practices like unit testing and integration test in software engineering. The goal of Jenga is to collect a huge library of data corruptions that occur in the real world, and uses these to automate the testing of ML models, ideally with an integration into upcoming systems for continuous integration for ML [14].

In the future, we aim to extend Jenga to extend more diverse tasks (e.g., regression problems or ranking problems). We will continue to work on Jenga as part of our recently proposed vision for automated ML model monitoring with respect to data quality [15]. We hope that Jenga can contribute to future research on data governance for end-to-end management platforms for ML.

REFERENCES

[1] Steffen Bickel, Michael Brückner, and Tobias Scheffer. 2009. Discriminative learning under covariate shift. *JMLR* 10, 2137–2155.

[2] Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. 2018. Deep Learning for Missing Value Imputation in Tables with Non-Numerical Data. In *CIKM*. 2017–2025.

[3] Marcus D Bloice, Peter M Roth, and Andreas Holzinger. 2019. Biomedical image augmentation using Augmentor. *Bioinformatics* 35, 21 (2019), 4522–4524.

[4] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *SysML*.

[5] Yeounoh Chung, Tim Kraska, Steven Euijong Whang, and Neoklis Polyzotis. 2018. Slice finder: Automated data slicing for model interpretability. *SysML*.

[6] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* (2008).

[7] Jiayuan Huang, Arthur Gretton, Karsten M Borgwardt, Bernhard Schölkopf, and Alex J Smola. 2007. Correcting sample selection bias by unlabeled data. *NeurIPS*, 601–608.

[8] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2020. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *ICDE*.

[9] Zachary C Lipton, Yu-Xiang Wang, and Alex Smola. 2018. Detecting and Correcting for Label Shift with Black Box Predictors. *ICML*.

[10] R. J. A. Little and D. B. Rubin. 2002. *Statistical analysis with missing data. 2nd ed.* Wiley-Interscience, Hoboken, NJ,.

[11] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. *SOSP*, 1–18.

[12] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Record* 47, 2, 17.

[13] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. 2019. Failing loudly: an empirical study of methods for detecting dataset shift. In *NeurIPS*. 1394–1406.

[14] Cedric Renggli, Frances Ann Hubis, Bojan Karlaš, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Ease. ml/ci and Ease. ml/meter in action: towards data management for statistical generalization. *VLDB* 12, 12 (2019), 1962–1965.

[15] Tammo Rukat, Dustin Lange, Sebastian Schelter, and Felix Biessmann. 2019. Towards Automated ML Model Monitoring: Measure, Improve and Quantify Data Quality. *ML Ops workshop at MLSys* (2019).

[16] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Engineering Bulletin* 41.

[17] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *PVLDB* 11, 12, 1781–1794.

[18] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1781–1794.

[19] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. 2020. Learning to Validate the Predictions of Black Box Classifiers on Unseen Data. *SIGMOD*.

[20] D Sculley et al. 2015. Hidden technical debt in machine learning systems. *NeurIPS*, 2503–2511.

[21] Michael Stonebraker and Ihab F Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.