Software Framework for Data Error Injection to Test Machine Learning Systems

Jukka K. Nurminen, Tuomas Halvari, Juha Harviainen, Juha Mylläri,
Antti Röyskö, Juuso Silvennoinen, and Tommi Mikkonen
Department of Computer Science, University of Helsinki, Finland
Email: first.[initial.]last@helsinki.fi

***Abstract*—Data-intensive systems are sensitive to the quality of data. Data often has problems due to faulty sensors or network problems, for instance. In this work we develop a framework to emulate errors in data and use it to study how machine learning (ML) systems work when the data has problems. Our goal is to show how data errors can be emulated and how that can be used to study the behavior of ML solutions.**

I. Introduction

Data-intensive systems are sensitive to the quality of data. In deployed systems, data frequently has problems: sensor readings are drifting, network connection problems create gaps in IoT data, and so on. To ensure that systems are working correctly in different conditions, we need to improve our understanding how different problems in input data influence predictions and other results of systems. Especially in machine learning (ML) systems, we face questions that not only influence the testing phase but also the development decisions. Such questions include the following:

– Should we train the system with perfect or with erroneous data? Examples of erroneous data are far less common than examples of correct data but we may still have a good understanding about the kinds of problems in data the system will face over its lifetime.
– Are some ML algorithms, architectures, or hyperparameter selections more robust towards errors in data than others?
– How trustworthy the results of the algorithms are when used in real-life settings, which include errors in input data?

The difficulty of making a system deal with data-errors comes from multiple sources. To begin with, errors come in different forms. Some of them are systematic (e.g. sensor drift), whereas others are more random (e.g. a broken network connection). They happen infrequently so in the training material there may not be many examples of erroneous behavior. Furthermore, it is not obvious what we should do to deal with errors – change the associated training data, change the model, or simply ignore the erroneous output somehow.

Unfortunately, testing how a system behaves with different kinds of errors in data has been difficult. Especially the rare cases of multiple errors at the same time is hard to test with real measured data. To solve related problems, the importance of errors in data, ”dirty data” has been recently observed. E.g. Qu et al. [1] investigate how dirty data influences the operation of a set of classification, clustering, and regression algorithms. A number of tools e.g. [2] have been developed to find and fix problems in datasets. While these tools allow fixing problems in datasets there does not seem to be tools which allow generation of problems to the datasets. Adversarial ML [3] aims to generate cases which make the ML model behave erroneously. The idea is to check how an adversarial opponent can cause problems to the systems. The main idea of our implementation is not to consider an explicitly malicious adversary but to generate datasets which encompass typical problems in data.

The approach we are suggesting is to use an artificial error generator, which modifies the input data according to some rules. These rules, which form the error model, should characterize the typical problem situations the system is likely to encounter. We can then use the modified data to test the operation of the system in the faulty conditions, or we can use the faulty data in training to aim for more robust systems.

In this paper, we propose an approach to artificially generate errors to input data to test and to improve the training of ML systems. As a technical contribution, we describe our implementation of a prototype system dpEmu for the task. Finally, we show with an example what kind of observations the system allows.

The rest of this paper is structured as follows. In Section II, we discuss related work. In Section III, we introduce dpEmu, its architecture and rationale, and key use cases. In Section IV we present a concrete example where errors are generated into an object detection dataset. In Section V we draw some final conclusions.

II. Background and Related work

As observed by Breck et al. [4] “Software testing is well studied, as is machine learning, but their intersection has been less well explored in the literature”. Recently, however, the situation has started to change. As evidenced by recent surveys [5], [6] researchers have realized the importance of testing in the development of artificial intelligence (AI), including in particular machine learning (ML) systems. The key observation is that ML systems cannot be tested in the same way as classical software systems. Instead, new approaches are needed.

Focusing on data quality for testing is one such approach. Qu et al. [1] suggest a set of guidelines for algorithm selection and data cleaning based on their evaluation of classification, clustering, and regression algorithms. They also observe that unnecessary cleaning of dirty data can be wasteful and therefore understanding what is appropriate cleaning and appropriate algorithm in different cases is important. To enable this they encourage the development of models of how error types and rates as well as data size and algorithm influence the performance. Our framework is one way to allow such models be created and analyzed.

A number of tools e.g. [2] have been developed to find and fix problems in datasets. Li et al. [7] have developed a benchmark, CleanML, to investigate the effect of data cleaning and algorithm selection. They observe than data cleaning alone does not improve performance and can even be harmful. Instead selection of proper model is important for cleaning approaches to effective. These observations seem to highlight the importance of experimenting with alternative algorithms and data cleaning approaches in the development of ML systems. An explicit goal of the dpEmu system is to make such exploration easier.

A number of papers investigate fault injection. They are typically focused on embedded systems and looking at ways for software to allow recovery of hardware problems (see e.g. [8]). Our work is related, but instead of system hardware being the source of the problem, our faults arise of problems in external data collection and manipulation systems and are reflected as problems in the data that the system is using.

A system close to our thinking is AVFI [9], which is used for injecting faults to systems controlling autonomous vehicles. Its focus is on system level analysis of autonomous vehicle behavior and, similarly to our work, it allows studying how different sensor problems influence the operation.

DeepRoad [10] uses GAN to artificially generate various weather conditions to road scenes. It is intended for testing autonomous driving systems. It found thousands of problems in state-of-the-art self-driving systems highlighting the importance of thorough testing of these systems. DeepRoad focused on changes caused by nature to the weather and to the object detection software while our prime target has been changes to data because of the technical problems or inadequacies.

DeepMutation [11] is another systems close to our thinking. In DeepMutation the system generates artificial faults to both data and model implementation. Many of the mutation operators they suggest are similar to ours.

III. Towards Data Problems Emulation: dpEmu

ML systems are typically developed with pipelines. Data acquisition is followed by data cleaning, preprocessing, training the model, and evaluating the result. The steps can vary and be more detailed depending on the case. To establish the data error injection as part of the development process we feel tools, which allow it to be merged, to the development pipeline are essential. This observation was the starting point of our work towards a software framework. Its goals were the following (Fig. 1):

– Easy and flexible modelling of the types of data faults the system is likely to encounter.

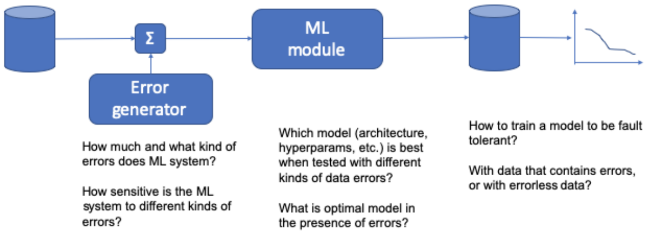

Σ
ML module
Error generator
How much and what kind of errors does ML system?
How sensitive is the ML system to different kinds of errors?
Which model (architecture, hyperparams, etc.) is best when tested with different kinds of data errors?
What is optimal model in the presence of errors?
How to train a model to be fault tolerant?
With data that contains errors, or with errorless data?

Fig. 1: Conceptual view of the system with examples of questions that the system is able to answer.

– Parameterization of the data fault generation so that developers can study how sensitive their systems are to different kinds of faults and what are the thresholds of data problems when the system starts to loose its performance.
– Visualization of the results with different error models and parameters.
– Bookkeeping to allow going back to the sources of problems.
– Embedding the error injection and error visualization to the development pipeline.
– Integration of data problem emulation to different development pipelines.

To satisfy these goals, we have created a generator framework for emulating data problems, called dpEmu. The framework is able to generate errors in your training and/or testing data in a controlled and documentable manner, and it enables emulating data problems in the use and training of ML systems as depicted in Fig. 2. The Runner routine introduces errors in a dataset, following the definitions set in the Error definition tree. Then, the resulting data is prepossessed and run by ML models, which produces final results that can be visualized by the user.

DpEmu can run one or more ML models on any data using different values for the error parameters and visualize the results. Written in Python, dpEmu can easily interact with any Python based ML framework such as Sklearn, TensorFlow, or PyTorch. It is is available as open source at https://github.com/dpEmu/dpEmu.

A. Error Tree

Error generation in dpEmu is based on error generation trees, which consist of tree nodes. The most common type of leaf node is Array, which can represent a Numpy array (or Python list) of any dimension. Even a scalar value can be represented by an Array node provided that node is not the root of the tree. If the key unit of the data set is a tuple (like .wav audio) a Tuple node should be used as the leaf.

The simplest and most commonly used non-leaf node type is Series. For example, one might choose to represent a matrix of data as a series of rows. In that case one would create an Array node to represent a row, and provide it as the argument to a Series node:

```
row_node = Array()
```

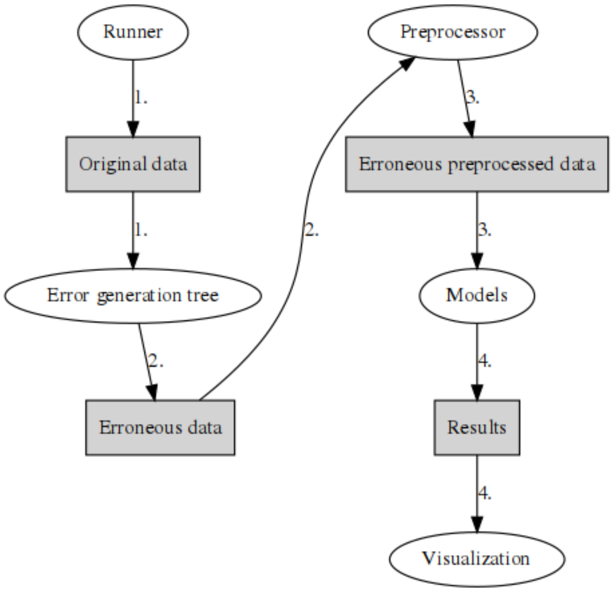


Fig. 2: DpEmu pipeline.

```
root_node = Series(row_node)
```

A TupleSeries represents a tuple where the first dimensions of the tuple elements are ”the same”. For example, if we have one Numpy array, say X, containing the input data, and another, Y, containing each data point’s correct label, we may choose to represent (X, Y) as TupleSeries. In general, there are usually more than one valid way to represent the structure of the data as a tree. For example, a two-dimensional Numpy array can be represented as a matrix, i.e. a single Array node; as a list of rows, i.e. a Series with an Array as its child; or as a list of lists of scalars, i.e. a Series whose child is a Series whose child is an Array.

Filters, which act as the error sources, can be added to leaf nodes. Dozens of filters, such as Snow, Blur and SensorDrift are provided out of the box. They can be used to manipulate data, including images, time series and sound. Users can also create their own custom error sources by subclassing the Filter class. To create a filter, one should call the constructor and provide string identifiers for the error parameters of that filter. To attach the filter to a leaf node, the node’s addfilter method is called with the filter object as the parameter.

Once the error generation tree including the desired filters is complete, generate_error method of the root node of the tree will modify the original input data as specified by the error generation tree. The resulting dataset, with injected errors, is then used as new input data. The method takes two arguments, the data into which the errors are to be introduced, and a dictionary of error parameters based on which the errors are generated.

As an example, let us consider the MNIST dataset (https://www.openml.org/d/554) of handwritten digits. The input consists of 70000 rows where each row is a 784 pixel (i.e. 28×28) black and white image of a handwritten digit (Fig. 4a). The next step is to choose the node type for error generation. Since the inputs are an indexed collection of images, it is natural to represent them as a series of arrays, each array corresponding to a single image. We can now add one or more error sources. Error sources are known as Filters in dpEmu parlance, and they can be attached to Array nodes.

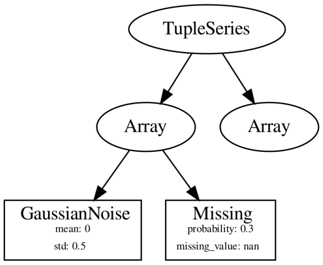
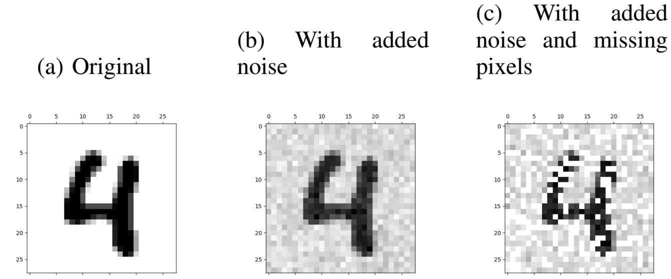


Fig. 3: Visualization of a possible error generation tree with added gaussian noise and missing pixels.



(a) Original

(b) With added noise

(c) With added noise and missing pixels

Fig. 4: Sample MNIST figure processed with filters.

```
image_node = Array()
series_node = Series(image_node)
image_node.addfilter(GaussianNoise(”mean”, ”
    std”))
```

The GaussianNoise Filter adds noise drawn from a Normal distribution. The constructor takes two String arguments, which are identifiers for the parameters. We will also have to provide the values of these parameters and actually generate the errorified data:

```
params = {”mean”: 0.0, ”std”: 20.0}
err_data = series_node.generate_error(data,
    params)
```

The same image with added noise is provided in Fig. 4b.

We are not limited to one error source (i.e. Filter) per node. The Missing Filter takes each value in the array and changes it to a value of our choice with the probability of our choice:

```
image_node.addfilter(Missing(”probability”, ”
    missing_value_id”))
params = {”mean”: 0.0, ”std”: 20.0, ”
    probability”: .3, ”missing_value_id”: np.
    nan}
err_data = series_node.generate_error(data,
    params)
```

The same image with added noise and missing pixels is presented in Fig. 4c.

B. Exploratory Execution

To support exploratory use, dpEmu includes a runner system, or simply Runner. It creates subprocesses for each set of error parameters, and in each subprocess all of the included ML models are run, with possibly multiple sets of hyperparameters. This allows distributing the processing. When all subprocesses are finished running, the system returns the results as a pandas DataFrame object which can then be used for visualizing.

The runner takes the following as input when it is run: train data, test data, a preprocessor class, preprocessor parameters, an error generation root node, a list of error parameters and a list of ML models and their parameters:

```
df = runner.run(
    train_data=train_data,
    test_data=test_data,
    preproc=Preprocessor,
    preproc_params={},
    err_root_node=get_err_root_node(),
    err_params_list=get_err_params_list(),
    model_params_dict_list=
        get_model_params_dict_list()
)
```

The list of error parameters is simply a list of dictionaries, which contain the keys and error values for the error generation tree. The list of ML model parameters is a list of dictionaries containing three keys: "model", "params_list" and " use_clean_train_data ". The value of "model" is the name of a class where the actual model is called. The value of "params_list" is a list of dictionaries where each dictionary contains one set of parameters for the model. The model is run with all of these parameter combinations in each subprocess. If " use_clean_train_data " is true, Runner will always pass the original, clean train data to the model in addition to the errorified test data. If there is no train data, a None value can also be passed to the Runner.

User defined preprocessor is run twice in every subprocess, right after the error generation, using both clean and errorified train data so that the correct version of the train data can be passed to each model. The preprocessor implements a function run(train_data , test_data , params), and it returns the preprocessed train and test data. The preprocessor can return additional data as well, and it will be listed as separate columns in the DataFrame which the runner returns:

```
class Preprocessor:
    def run(self, train_data, test_data,
        params):
        return train_data, test_data, {}
```

Each model class should implement function run(train_data , test_data , parameters) which optionally trains the model on the training data and tests the model with test data with given model parameters and returns a dictionary containing the scores and possibly additional data to be added to the resulting DataFrame:

```
class Model:
    def run(self, train_data, test_data,
        params):
        return {}
```

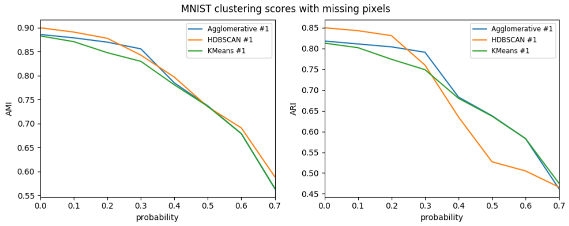


Fig. 5: Visualization of AMI and ARI scores for three models when clustering MNIST dataset with missing pixels at different error levels. The scores for HDBSCAN are hyperparameter-optimized.

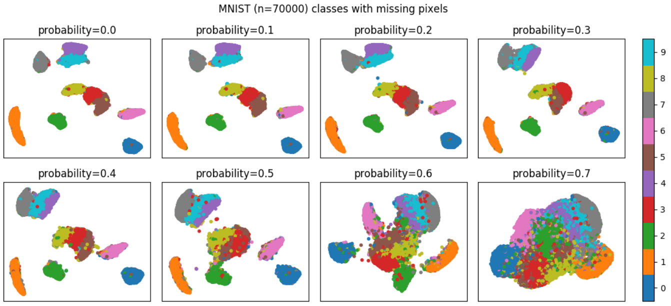


Fig. 6: 2D visualization of the MNIST dataset with different error probability levels.

All of the data given to the specific model has first passed through error generation and preprocessing. You can for example use the preprocessor to write the errorified data to files and then call the CLI of a ML model using our run_ml_module_using_cli(cline), which returns the output of an external model to be parsed.

C. Visualization

In addition to error generation, dpEmu can be used for visualizing the results for the desired parameter combinations. It supports visualizing (i) hyperparameter-optimized scores for each model at different error levels (Fig. 5); (ii) interactive plots where data points can be clicked to visualize it; (iii) 2D visualization of the dataset at different error levels using original labels (Fig. 6); (iv) the model parameter values which give the best results; (v) interactive confusion matrices of the classification results (Fig. 7); and (vi) the error generation tree.

D. Use Cases

To test the usefulness of dpEmu we have studied a number of use cases with different kinds of data and algorithms. These studies include the following:

– Comparing some clustering algorithms' performance when clustering images from datasets like MNIST or Fashion MNIST while adding different amounts of error,

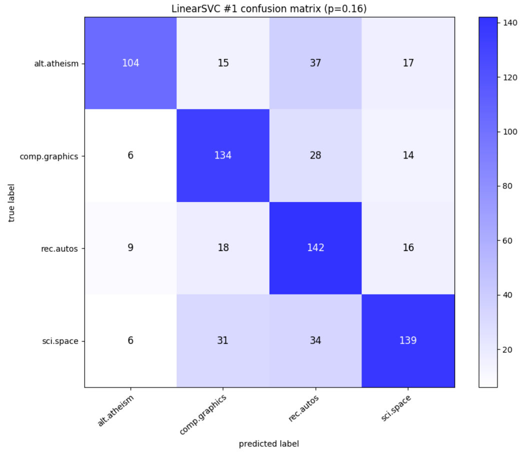


Fig. 7: Visualization of a confusion matrix for a LinearSVC model from a 20 Newsgroups dataset with areas.

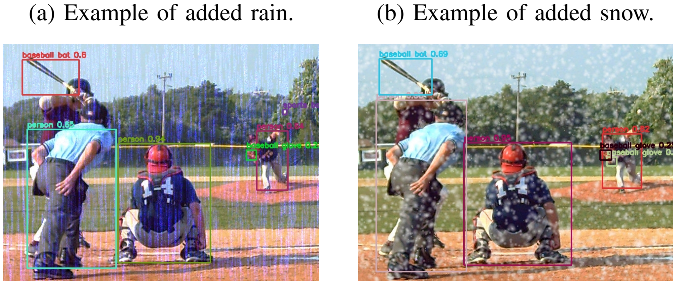
for example gaussian noise or missing pixels, to the images. Dimensionality reduction is applied to the images in the preprocessing phase before clustering. The evaluation is done by calculating AMI and ARI scores using the original labels provided. Also in case of algorithms like HDBSCAN, optimal hyperparameters at varied error levels are also studied.

- Comparing different classification algorithms' performance when classifying texts from the 20 newsgroups dataset while training the models with both clean and dirty data at different error levels. Error sources such as OCR errors and random missing areas are used. Optimal hyperparameters at varied error levels are also studied.
- Forecast future values of time series using the LSTM model when measured values have (i) arbitrary errors; (ii) systematic drift, and (iii) gaps.
- Object detection, which we will discuss in the next section in detail.

Results for different use cases will be available at https://github.com/dpEmu/dpEmu/tree/master/results. Some are also available as Jupyter notebooks at https://github.com/dpEmu/dpEmu/tree/master/notebooks.

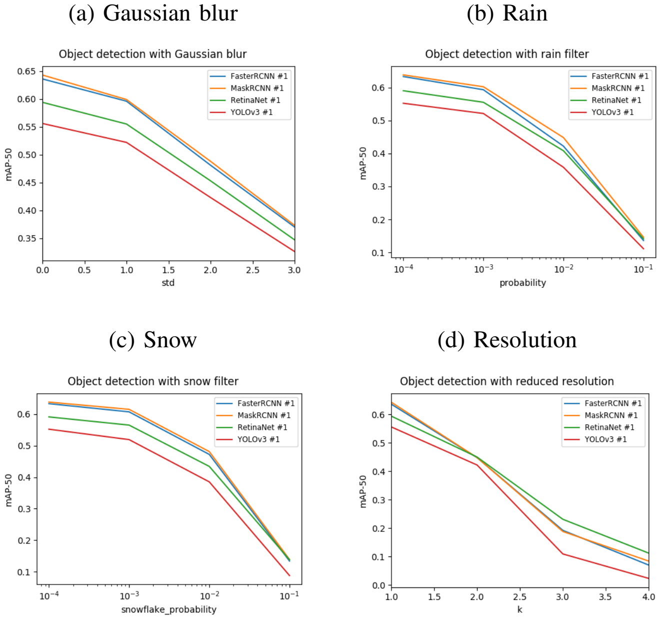
IV. Object Detection Case Study

As a concrete example, we next use dpEmu to analyze the accuracy of different object detection models. The code and more details of the analysis is available at https://github.com/dpEmu/dpEmu/blob/master/demo/Object_detection_case_study/Object_Detection_case_study.md. In the study, we compared the performance of three models from FaceBook's Detectron project (FasterRCNN, MaskRCNN and RetinaNet) and YOLOv3 model from Joseph Redmon, when different error sources were added. We used 118 287 jpg images (COCO train2017) as train data and 5000 jpg images (COCO val2017 http://cocodataset.org/#download) as test data to calculate the mAP-50 scores. We used the pretrained weights for FasterRCNN (e2e_faster_rcnn_X-101-64x4d-FPN_1x), MaskRCNN (e2e_mask_rcnn_X-101-64x4d-FPN_1x) and RetinaNet (retinanet_X-101-64x4d-FPN_1x) from Detectron's model zoo. YOLOv3's weights were trained by us, using the Kale cluster of University of Helsinki. The training took approximately five days when two NVIDIA Tesla V100 GPUs were used.


(a) Example of added rain.

(b) Example of added snow.

Fig. 8: Examples of rain and snow filters.


(a) Gaussian blur

(b) Rain

(c) Snow

(d) Resolution

Fig. 9: Effect of different filters to the accuracy of object detection algorithms.

For dpEmu, we have defined a number Filters to generate different kinds of errors in images, including Gaussian blur, Added rain, Added snow, and Resolution change. The Gaussian blur filter added normally distributed error with mean 0 to the data. The standard deviation parameter was varied. Added rain and snow filters introduced simulated rain and snow to images. Sample effects of the Added rain and snow filters can be seen in Fig. 8. The resolution was changed with the formula:

$$\text{image}[y][x] = \text{image}\left[k\left\lfloor\frac{y}{k}\right\rfloor\right]\left[k\left\lfloor\frac{x}{k}\right\rfloor\right]$$

The results of the object detection are presented in Fig. 9. As expected, the figure shows that object detection accuracy drops when the amount of error in the picture increases. Because it is difficult to compare the severity of different kinds of errors (the x-axes), it is difficult to state which type of error is the most harmful. However we can see that with Gaussian blur, Added rain, and Added snow (Fig. 9a, 9b, 9c), the order of the different models remains the same, and MaskRCNN is the most accurate with all three error types. With Added rain, FasterRCNN accuracy drops faster than the accuracy of MaskRCNN.

Interestingly, with reduced resolution (Fig 9d), the accuracy order changes as the resolution gets worse. RetinaNet, which initially was 3rd, gave more accurate results than FasterRCNN and MaskRCNN, which initially were clearly more accurate. These results are tentative and a more extensive analysis would be useful to evaluate the merits of different models in changing conditions. They are, however, demonstrators of the kinds of analysis dpEmu supports.

V. Conclusions

With the aim to develop robust and trustworthy ML systems we have created dpEmu to encourage developers to evaluate how their models and systems work when system input data has problems. The system can be used for multiple purposes, such as investigating how a trained model or a entire system tolerates different kinds of errors in its input data; studying which model and hyperparameterization is the best when input data has certain kinds of errors or how alternative data cleaning approaches influence the operation of resulting model; evaluating trade-offs between model accuracy versus model robustness; and quantifing the accuracy difference when the model is trained with error-free or fault injected data. At present, dpEmu still is a prototype and as usual it is not perfect in terms of functionality and usability. However, it already acts as demonstrator regarding how robustness and tolerance to data errors can be integrated to the development pipeline of ML systems.

Popular ML libraries, such as Sklearn or TensorFlow, have extensive collections of functions for evaluating the models as well as splitting the datasets for training and testing parts. However, none of these, seem to have built-in support for studying the model behavior in case of erroneous input data. DpEmu can be used together with these libraries to add one step prior to the actual training of the model. The addition of one more step however will increase the training effort a lot. In addition to the actual training it is possible to have another training loop which searches for the best possible architecture and hyperparameterization for the system [12]. Introduction of a third loop with different data errors to ensure the system works in optimal, or adequate, fashion also with data errors will further increase the already massive computational effort.

To ensure that ML models, and systems based on those, are robust it is important to test how the systems work when input data has errors. We envision that in the future models of typical errors would exists for all regularly used data, such as for the behavior of different kind of sensors and their connectivity. Such models already exist for e.g. problems in textual document for OCR [13]. The developers would then choose among different predefined error models and parameterize them to match their needs. If a ready-made error model is missing, users could create their own. For their maintenance, ML systems could not only track their behavior as they do now but also track the behavior of the error model that was used in their development. A deviation from the error model behavior could be an indication of the need to reconsider if the model in use is still optimal for the newly observed error behavior.

References

[1] Z. Qi, H. Wang, J. Li, and H. Gao, "Impacts of Dirty Data: and Experimental Evaluation," 3 2018. [Online]. Available: http://arxiv.org/abs/1803.06071

[2] N. Hynes, D. Sculley, G. Brain, M. T. Google Brain, and M. Terry, "The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets," in *NIPS MLSys Workshop*, 2017. [Online]. Available: https://github.com/brain-research/data-linter

[3] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proceedings of the 4th ACM workshop on Security and artificial intelligence - AISec '11*. New York, New York, USA: ACM Press, 2011, p. 43. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2046684.2046692

[4] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "What's your ML Test Score? A rubric for ML production systems," 2016.

[5] H. B. Braiek and F. Khomh, "On Testing Machine Learning Programs," 12 2018. [Online]. Available: http://arxiv.org/abs/1812.02257

[6] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," 6 2019. [Online]. Available: http://arxiv.org/abs/1906.10742

[7] P. Li, X. Rao, J. Blase, Y. Zhang, X. Chu, and C. Zhang, "CleanML: A Benchmark for Joint Data Cleaning and Machine Learning [Experiments and Analysis]," 4 2019. [Online]. Available: http://arxiv.org/abs/1904.09483

[8] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 5 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/6850649/

[9] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "AVFI: Fault Injection for Autonomous Vehicles," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 6 2018, pp. 55–56. [Online]. Available: https://ieeexplore.ieee.org/document/8416212/

[10] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. New York, New York, USA: ACM Press, 2018, pp. 132–142. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3238147.3238187

[11] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepMutation: Mutation Testing of Deep Learning Systems," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. 2018-October. IEEE Computer Society, 11 2018, pp. 100–111.

[12] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving Deep Neural Networks," *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312, 1 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128154809000153

[13] H. S. Baird, "The State of the Art of Document Image Degradation Modelling," in *Digital Document Processing*. Springer, London, 2007, pp. 261–279. [Online]. Available: http://link.springer.com/10.1007/978-1-84628-726-8_12