Evaluating Deep Neural Networks in Deployment: A Comparative Study (Replicability Study)

Eduard Pinconschi
epincons@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Divya Gopinath
divya.gopinath@nasa.gov
KBR Inc., NASA Ames
Mountain View, California, USA

Rui Abreu
rui@computer.org
INESC-ID, University of Porto
Porto, Portugal

Corina S. Păsăreanu
pcorina@andrew.cmu.edu
Carnegie Mellon University, NASA Ames
Mountain View, California, USA

Abstract

As deep neural networks (DNNs) are increasingly used in safety-critical applications, there is a growing concern for their reliability. Even highly trained, high-performant networks are not 100% accurate. However, it is very difficult to predict their behavior during deployment without ground truth. In this paper, we provide a comparative and replicability study on recent approaches that have been proposed to evaluate the reliability of DNNs in deployment. We find that it is hard to run and reproduce the results for these approaches on their replication packages and even more difficult to run them on artifacts other than their own. Further, it is difficult to compare the effectiveness of the approaches, due to the lack of clearly defined evaluation metrics. Our results indicate that more effort is needed in our research community to obtain sound techniques for evaluating the reliability of neural networks in safety-critical domains. To this end, we contribute an evaluation framework that incorporates the considered approaches and enables evaluation on common benchmarks, using common metrics.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software testing and debugging**.

Keywords

Trustworthy AI, Testing, Neural Networks.

ACM Reference Format:
Eduard Pinconschi, Divya Gopinath, Rui Abreu, and Corina S. Păsăreanu. 2018. Evaluating Deep Neural Networks in Deployment: A Comparative Study (Replicability Study). In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

ISSTA 2024, 16-20 September, 2024, Vienna, Austria

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Deep Neural Networks (DNNs) have found many applications in safety-critical domains [12, 18, 20] raising important questions about their reliability. Even highly trained, highly accurate DNNs are not 100% accurate, and can thus contain errors which can have costly consequences in domains such as medical advice [12, 18], self-driving cars [18], or banking [20]. However, detecting these errors during deployment is difficult due to the absence of ground truth at inference time.

In this paper, we aim to provide a comparative and replicability study on recent techniques that were published in the software engineering community that aim to evaluate the reliability of DNNs in deployment. We restrict our attention to white-box, post-hoc methods, that take a pre-trained model, perform white-box analysis of the internals of the model (possibly guided by labeled data), and produce monitors to be deployed at run time, with the goal of predicting mis-classifications. After a preliminary literature review, we have identified two recent, state-of-the art approaches on this topic: SelfChecker [32] and DeepInfer [10]. SelfChecker computes the similarity between DNN layer features of unseen instances and the samples in the training set, using kernel density estimation (KDE) to detect mis-classifications by the model in deployment. DeepInfer infers data preconditions from the DNN model and uses them to determine the model's correct or incorrect prediction. In addition, we propose to use Prophecy [15] for our study. Although it was not used before for determining the reliability of outputs, Prophecy bears similarities to both DeepInfer and SelfChecker, as it also aims to generate data preconditions (as in DeepInfer) albeit at every layer, driven by training data (similar to SelfChecker).

We first try our best to reproduce DeepInfer and SelfChecker based on their original artifacts, i.e., tabular datasets and models for DeepInfer vs. image datasets and models for SelfChecker. After that, we go one step further to empirically compare these approaches against each-other's datasets and models. Experimental results indicate that the approaches, although very recent and hence well maintained, are difficult to reproduce on their original artifacts; it is even more difficult to run these approaches on models and datasets other than their own. Further, for the cases where they can be successfully run on the same artifacts, it is difficult to compare them due to lack of clear metrics for measuring performance. Our results highlight that significant effort should be spent by our community to develop sound techniques for evaluating the reliability of deep

neural networks in deployment. To this end, we propose a common, public framework, TrustDNN, that we built for evaluating and comparing the different techniques, using common metrics.

The framework incorporates robustified versions of DeepInfer and SelfChecker as well as Prophecy, together with carefully curated datasets and models. Using this framework, we run a comparative evaluation of the three approaches on tabular and image data and the respective models. We find that DeepInfer generally performs the best on both tabular data and image data (after our modification). Further, Prophecy is the only approach that can handle both image and tabular data without modifications, while SelfChecker appears to have the highest resource consumption. Overall, the results indicate that the three considered approaches have complementary strengths. Our framework provides a portfolio approach to address the hard problem of trustworthy AI. To summarize, we make the following contributions;

- We identify state-of-the-art approaches (SelfChecker [32], DeepInfer [10]), that evaluate reliability of DNN models and are available as open-source tools. These approaches have commonality in their workflow but are sufficiently different in their methodologies to warrant comparison.
- We present the first-time application of Prophecy [15] to evaluate the reliability of DNN models.
- We develop tools, *TrustBench* and *TrustDNN*, a unified and standard framework to curate benchmarks and tools for replication and comparative studies.
- We present a replication study that goes beyond mere re-running of the tools; we (i) enlist all issues encountered and possible solutions, (ii) evaluate existing approaches on new domains, (iii) *refactor and extend* existing approaches to enable re-use and application to new domains.
- We present a comparative study of three approaches and discuss observations regarding effectiveness and efficiencies on benchmarks from domains with image and tabular data.
- We have made publicly available the replication package along with artifacts and results (https://zenodo.org/doi/10.5281/zenodo.12803632).

2 Approaches

In this section, we briefly survey approaches that measure the reliability of machine learning models. Our purpose is not to perform a comprehensive survey of all the possible approaches in this space but to identify the most recent, well-maintained, state-of-the-art studies that can be replicated. To this end, we selected techniques based on the following criteria;

- Techniques that can be considered state-of-the-art: They have been proposed in the past three years or have considerable citations.
- Techniques that have been fully developed and made publicly available, and can be quantitatively evaluated.
- Approaches that show potential to have broad applicability (i.e. are not designed to work only for specific input domains).
- Techniques which have not already been thoroughly evaluated and shown to perform poorly in comparison with other approaches.

Use of confidence scores based on softmax probabilities of DNN classifiers [17], and information theory metrics such as entropy, have been studied by a number of previous work and shown to be unreliable [30], [13], [14].

The work in [19] uses nearest neighbor classifiers to measure model confidence. However, subsequent work [13] evaluated this approach to be less efficient and not effective on large datasets and complex models. This work [13] proposed an approach called ConfidNet, which builds a ConvNet model (on top of the an existing pre-trained model) to learn the confidence criterion based on the true class probability for failure prediction. More recent work [32] performs a quantitative comparison with ConfidNet to highlight its ineffectiveness when the training data has very few mis-classified instances. Dissector, proposed by Wang et al. [31], focuses on distinguishing inputs representing unexpected conditions from normal inputs by training a number of sub-models. However, it is applicable mainly for image models. This tool was also evaluated by the more recent work [32], which highlights that the process of building sub-models is manual and highly time-consuming. SelfOracle [26] is a technique that monitors video frames or image sequences to identify unsupported driving scenarios based on the estimated model confidence being lesser than a threshold. The approach is specifically designed for self-driving car models and its performance has been evaluated to be poor on other DNN types [32]. Recent work [10] further points out that SelfOracle, ConfidNet, and Dissector are not applicable to models processing numeric data.

Based on this overview, we identified two existing approaches for our study; SelfChecker [32]; first proposed in 2021, with more citations than other techniques and with a publicly available tool that was shown to be better than previous approaches, and DeepInfer [10]; very recent work due to appear in ICSE 2024 with a publicly available tool. The above two techniques have been individually evaluated only on image (SelfChecker) and tabular data (DeepInfer) respectively but the design is not restrictive to one particular domain. We also include another tool, Prophecy [15] in our study. This tool mines assume-guarantee type rules from pre-trained models and we explore the application of the tool for the first time to determine the reliability of models. We believe it holds potential since the approach is agnostic to the type of inputs.

Figure 1 presents an overview of the workflow of the three aforementioned approaches. Each approach typically has an offline analysis phase, which takes as input a pre-trained DNN model, data labelled with ground-truth such as the training and validation datasets, and applies an algorithm to capture the information learned by the model during training. The outputs of this phase are artifacts that can be used to monitor model behavior at runtime. The model is accordingly updated with runtime checks before deployment. At inference time, when the model is exposed to inputs (unseen during training), each approach applies the checks in an attempt to determine if the output produced by the model can be trusted. There are notifications regarding the reliability of the model output and in some cases an advice regarding the correct output. Given that the monitors are built guided by the training profile of the model, they are applicable mainly to predict model behavior on unseen inputs that follow the distribution of the data that the model has been exposed to during training.

We describe below the three approaches in detail:

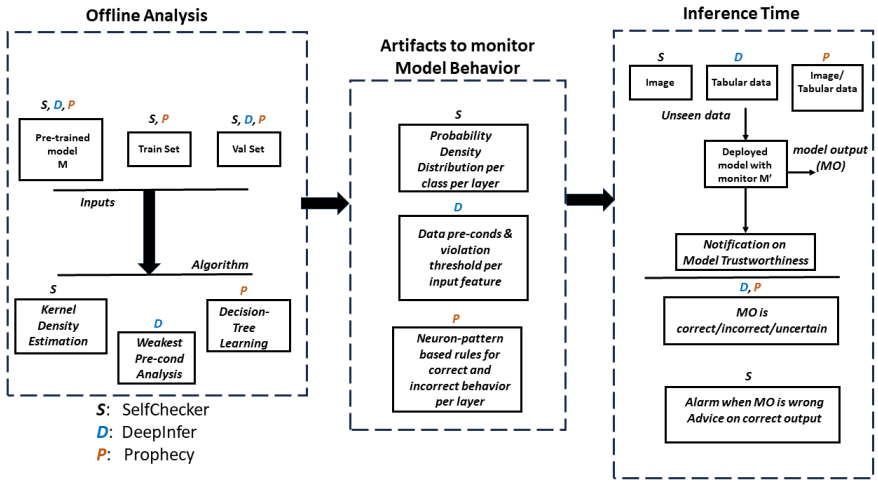

Figure 1: Workflow Overview to infer reliability of model behavior.

SelfChecker: The insight driving this approach is that DNN models typically reach a correct prediction before the final layer, and in some cases, the final layer may change a correct internal prediction into an incorrect prediction. The features extracted by the internal layers of a model contain information that can be used to check the correctness of the model output. Given a pre-trained model and the training data, the approach applies kernel density estimation (KDE) to determine the probability density distributions (PDD) of each layer's output. The PDD corresponding to each class is determined after every layer of the model.

At inference time, for a given input, SelfChecker estimates its probability density for each class within each layer from the computed density functions. The class for which the input instance induces the maximum estimated probability density, is considered as the inferred class for that layer. If a majority of the layers indicate inferred classes that are different from the model prediction, then SelfChecker triggers an alarm. The approach adopts a strategy to select a set of layers that positively contribute to the predication, by evaluating the performance of different combinations of layers on a validation set. A similar technique is adopted to determine the most probable correct prediction and the tool offers this as an advice. It also uses a boosting strategy based on the computed probable correct prediction to increase the alarm accuracy.

SelfChecker has been mainly evaluated on vision models, and on popular image datasets it has been shown to trigger correct alarms on 60.56% of wrong DNN predictions, and false alarms on 2.04% of correct DNN predictions. It has also been evaluated on self-driving car scenarios and has been able to achieve high F1-score (68.07%) in predicting behavior.

DeepInfer: This work attempts to circumvent the limitation of data-dependency which impacts the effectiveness of approaches such as SelfChecker; the training and validation datasets need to be representative of unseen instances. DeepInfer, on the other hand, applies a pure static analysis based technique, weakest pre-condition analysis using Dijkstra's Predicate Transformer Semantics on the DNN model to compute pre-conditions on input features. Starting with a post-condition for correct output; DNN output falling within a prediction interval, this analysis is used to compute conditions on the input of the output layer. This process is repeated iteratively until the input layer, to compute data preconditions. In order to handle multiple layers with hidden non-linearities, this work introduces a novel method for model abstraction and a weakest pre-condition calculus.

The pre-conditions represent the assumptions made by the pre-trained model on the input profile on which the model can be expected to behave correctly. The pre-conditions for every input variable or feature are evaluated against a validation dataset to obtain a mean value which acts as a threshold. At inference time, the unseen input is checked against the data pre-conditions and the violations are compared with the respective thresholds to predict if the model output is correct or incorrect. If sufficient evidence is not available to make a concrete decision, an "uncertain" output is produced.

The applicability, effectiveness and scalability of this approach is dependent on the architecture of the model, the precision of the abstraction and the input dimensionality. Unlike previous work, this technique has not been evaluated on vision models. It has been shown to perform well on multiple models with tabular data whose input dimensionality is much lower than images.

Prophecy: Given a DNN model F and an output property $P(F(X))$, Prophecy [15] extracts rules of the form, $\forall X : \sigma(X) \Rightarrow P(F(X))$, which can be viewed as *abstractions of model behavior*. $\sigma(X)$ is a pre-condition in terms of neuron patterns at the inner layers of the network; constraints on the values or activations (*on*, *off*) of a subset of neurons at one or more layers. $P(F(X))$ could be any property encoded as a constraint on the model output such as; a classifier's output being a certain label, the label being equal to ideal (correctness), the outputs of a regression model being within a certain range (safety) so on.

Previous work ([15], [22], [21], [28], [27], [29]), has explored the use of Prophecy in obtaining formal guarantees of behavior, repair, obtaining explanations, debugging and testing. We see potential in

using Prophecy to determine model reliability, which we explore in this work. The neuron-patterns based rules at a given layer of the model capture the logic (in terms of internal layer features). Given a set of passing and failing data, Prophecy could be used to mine rules corresponding to correct and incorrect model outputs, which in turn potentially capture the correct and vulnerable portions of the model respectively. Each rule has a formal mathematical form, which enables a quick and a precise evaluation on inputs. Further, the rules correspond to the logic learned by the model during training, irrespective of the nature or dimensionality of the input (image,tabular,text so on). Vision models typically comprise of an encoder portion which extracts features from the raw images and a head that uses the extracted features to make decisions. We envisage applying Prophecy to the dense layers following the encoder to capture this logic.

The offline analysis phase comprises of feeding Prophecy with a pre-trained model and a dataset labelled with passing and failing inputs. Prophecy uses decision-tree learning to extract rules corresponding to correct and incorrect model outputs in terms of the neuron values at every layer. At inference time, given an unseen input, it is evaluated against the set of rules for correct and incorrect output respectively at every layer. The rules at a given layer are mutually exclusive, therefore, the input either satisfies one of the rules for correct behavior, or one of the rules for incorrect behavior, and this corresponds to the decision as predicted by the rules at the layer. The correct/incorrect decision corresponds to the one that receives majority of votes. If there is a tie, the output is an "uncertain" decision.

2.1 Challenges

During our preliminary review of the approaches, we identified the following challenges for our study:

- *Lack of commonality in the input domains:* Existing approaches operate and have been evaluated on either image data (SelfChecker), or non-image data such as tabular data (DeepInfer). There is no common set of benchmarks (from different domains) on which tools measuring reliability could be evaluated on.
- *Ad-hoc and non-uniform data preparation methods:* We found that each tool adopts a different technique to obtain and process the data (train, test, validation datasets and models). It is essential to have a standard way to obtain the raw data and models, pre-process them and curate them for use.
- *Problems with setup:* We found a couple of issues with how the approaches are implemented, which impedes ease of use and generalizability. Many parameters are hard-coded; eg. the layers to be considered for a given model, the use of pre-computed values, and so on. The replication package of DeepInfer has different scripts for the same function for different models, has multiple folders containing duplicate information (models, data), which causes confusion and are also error-prone.
- *Lack of proper documentation:* There is very little documentation on the tool websites which clarifies the data preparation methods and the setup.
- *Lack of standard metrics for evaluation:* Discrepancies in evaluation methodologies and metrics across approaches complicate fair and compatible comparisons. Both the existing approaches we consider, make use of the confusion matrix metrics, namely true positive (TP), false positive (FP), true negative (TN), and false negative (FN). They use True Positive Rate (TPR), False Positive Rate (FPR), and F1-score to evaluate their techniques and compare them with others. However, there are differences in the way these metrics are actually computed in the respective codes, which makes it difficult to interpret if they have the same meaning. Further, the code that computes these metrics is closely coupled with the code implementing the functionality of the approach, which further hinders interpretation and also inhibits the application of other metrics to evaluate them.
- *Inherent difficulty in evaluating machine learning models:* Machine learning is a highly data-dependent process. Most of the approaches that are used to evaluate reliability of other models, themselves use data-driven algorithms. Therefore, in order to replicate results, we need the exact data used by the approaches in their analysis and inference phases. The inherent randomness in the training process of models can lead to varied outcomes. So even if we have access to the exact data, the results may still be different.

3 Methods

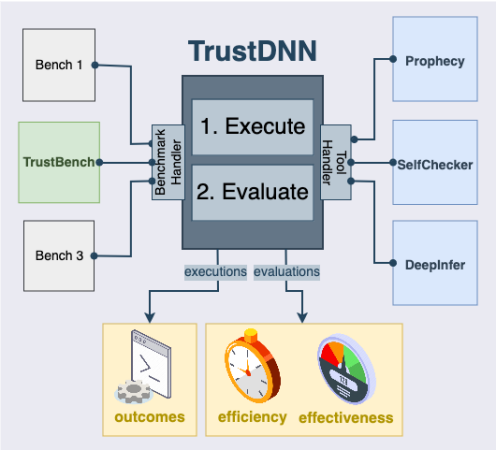

Figure 2: The TrustDNN Framework.

This section describes our methodology in performing this replicability and comparative study. We developed tools, namely *TrustBench* and *TrustDNN*, which seek to address the aforementioned challenges.

3.1 TrustBench

We designed TrustBench with Gray's definition of a good benchmark [16], considering it is repeatable, portable, scalable, representative, requires minimum changes in the target tools, and is simple to use. TrustBench is repeatable by automating the collection and preparation of datasets and models from different domains in a

standard manner. TrustBench ensures portability by using JSON files to define datasets, models, and other configurations, making it easy to share and replicate the setup across different systems. Additionally, its integration with the APIs of popular data sources like Kaggle [1] and Keras [2] enables it to fetch and use data consistently across various environments. TrustBench has a straightforward Command Line Interface (CLI) that features only three commands (collect, prepare, detail), making it easy to use. We made TrustBench representative regarding two previous tools [10, 32] by including their curated version of datasets and models. We present a detailed description of the process followed and issues encountered in section 3.3.1. TrustBench is scalable by allowing easy integration of new data sources by extending a class; refer to our public repository [3] for detailed instructions. Lastly, TrustBench does not require changes in the target tools as it outputs the datasets, models, and predictions at fine granularity and in a structured hierarchy.

3.2 TrustDNN Framework

A good framework is simple and quick to use, reusable in design and code, and provides enough adjustable and extensible features [25]. To compare various approaches in a unified set-up, we have developed the TrustDNN framework, illustrated in Figure 2. TrustDNN was built using Cement [4], a CLI framework that provides a robust interface/handler system. Leveraging that, TrustDNN offers a plugin system with dedicated handlers: a *ToolHandler* and *BenchmarkHandler*. These handlers facilitate the integration of new tools and benchmarks as plugins, each configurable through JSON files. This configurable and modular plugin system makes TrustDNN adjustable, extensible, and reusable in design and code.

The framework follows a two-step workflow (*execute* and *evaluate*), delivered through two commands, each with two actions, which makes it simple and quick to use. In the *execute* step, the *analysis* and *inference* phases are executed separately for each tool. The tools are expected to produce a file containing notifications as output after execution. Subsequently, during evaluation, the *efficiency* and *effectiveness* of the tools are measured by analyzing successful executions and their respective outputs. We provide more details on the usage of TrustDNN in our public repository [5].

The efficiency is typically measured in terms of time and memory consumption. To measure time efficiency, we register the duration of the process executing the tool. For memory consumption, we continuously monitor the Resident Set Size (RSS) of the process and its children, summed at intervals of 200 milliseconds. Following execution, we record the peak RSS. However, it's important to note that RSS only accounts for non-swapped physical memory usage and may not fully reflect all memory types, such as virtual memory.

The effectiveness of any algorithm making predictions, such as estimating the reliability of other models, measures how close the predictions are to the actual. We describe in detail the metrics used for this study in section 3.3.4.

[1] https://www.kaggle.com/ - the largest AI and ML community.
[2] Keras - https://keras.io/
[3] https://github.com/epicosy/TrustBench
[4] https://builtoncement.com/ - CLI framework for Python
[5] https://github.com/epicosy/TrustDNN

3.3 TrustBench and TrustDNN for our study

This section highlights the details specific to applying TrustBench and TrustDNN for this study.

3.3.1 **Data preparation using TrustBench.** The replication package of DeepInfer [9] provides artifacts for four case studies for tabular data, namely *Bank Marketing (BM)*, *German Credit (GC)*, *House Price (HP)*, and *PIMA Diabetes (PD)*. There are a total of 29 models corresponding to these case studies. The replication package contains a Data folder with a single dataset for each case study, representing the unseen or test data. A standard benchmark dataset for machine learning models and tools consists of train, validation and test data. Therefore, we employed TrustBench to prepare these datasets for the 4 case studies. After consultation with the authors of [10], we obtained the respective datasets from Kaggle and performed a standard train/validation/test split, with proportions specified by Xiao et al. [32] – allocating 80% for training and 10% each for validation and testing.

Bank Marketing (BM) [3]: To prepare this dataset, we referred to the code provided in the replication package [9] and consulted the most popular notebook associated with it on Kaggle [8], which follows a similar pre-processing approach. We followed the same steps by balancing each class to contain 5289 samples. However, this method is tailored to this binary scenario and may not be suitable for broader applications. Next, we employed categorical encoding for the columns representing *month*, *education*, and *outcome*. Subsequently, we utilized One-Hot Encoding via *get_dummies* function in Pandas, but this time individually for the remaining categorical fields (*job*, *marital*, *default*, *housing*, *loan*, *contact*) rather than applying it to the entire dataframe. Notably, a discrepancy arose in the one-hot encoded columns. Specifically, we obtained additional columns such as *job_admin*, *marital_divorced*, *default_no*, *housing_no*, *loan_no*, and *contact_cellular*. This inconsistency contradicts the expected behavior outlined in the documentation for the *get_dummies* function, which specifies the conversion of each variable into binary (0/1) variables corresponding to its unique values. Given this concern, we excluded those additional columns from our data preparation process to maintain compatibility with existing pre-trained models.

German Credit (GC) [4]: For preparing this dataset, we consulted the code provided in the replication package [9] and the notebook associated with the GC dataset that follows the most similar preprocessing steps [6]. Following the same approach, we normalized numerical columns to ensure a uniform order of magnitude and applied one-hot encoding to categorical columns. For normalization, we utilized the min-max scaler from *scikit-learn*[6] to scale the data between 0 and 1. The one-hot encoding process mirrored that used for the BM dataset. Additionally, the *Risk* column was transformed into a binary variable since it serves as the classification label. Interestingly, we observed in the GC preprocessing script within the replication package that the column *Saving accounts_rich* was dropped without explanation. We do the same, and following data preparation, we observe one additional column, *Purpose_vacation/others*, which we again excluded to maintain compatibility with existing pre-trained models.

[6] https://scikit-learn.org/stable/ - Python package for ML and data analysis

House Price (HP) [5]: For this dataset, we consulted the script in the replication package [9] and the notebook associated with it on kaggle [7] to apply the same min-max scaling as in the preparation of the GC dataset.

PIMA Diabetes (PD) [2]: For preparing this dataset, we performed our train/val/test split with the same random state 10 considered in the script provided in the replication package [9].

Table 1: Description of datasets (*#L* refers to labels count).

Dataset	#Samples				Input Size	#L
	Total	*Train*	*Val.*	*Test*		
BM [9]	2 116	NA	NA	2 116	28	2
GC [9]	200	NA	NA	200	22	2
HP [9]	292	NA	NA	292	10	2
PD [9]	153	NA	NA	153	8	2

(a) Tabular datasets: Original artifacts provided by DeepInfer [9]

Dataset	#Samples				Input Size	#L
	Total	*Train*	*Val.*	*Test*		
BM [3]	10 578	8 462	1 058	1 058	28	2
GC [4]	1 000	800	100	100	22	2
HP [5]	1 460	1 168	146	146	10	2
PD [2]	768	614	77	77	8	2

(b) Tabular datasets: Prepared using TrustBench.

Dataset	#Samples				Input Size	#L
	Total	*Train*	*Val.*	*Test*		
CIFAR [23]	60 000	40 000	10 000	10 000	32*32*3	10

(c) Standard Image dataset used by SelfChecker.

Table 1a contains details on the datasets in the original replication package of DeepInfer. Table 1b contains details on the datasets prepared using TrustBench for reuse by other approaches.

The replication package of SelfChecker [33] contains artifacts corresponding to only **CIFAR10**. This is a standard and popular benchmark [23] , which we obtain from Keras [1] and for which we follow the exact data preparation as performed by Y. Xiao et al. in their replication package [33].

3.3.2 **Models.** We use the 29 models provided in the replication package from [10] for tabular data. For the image data, we use the only model (ConvNet) provided in the replication package for SelfChecker [32]. Table 2 details the characteristics of the models used in this study and their accuracy on the respective test sets.

3.3.3 **Tools.** We examined the repositories for the tools, DeepInfer and SelfChecker, in order to integrate them into the TrustDNN framework. We found several problems as highlighted in the challenges; the repository for DeepInfer contains scripts specialized for each model and dataset and has un-necessary replication of code. We found SelfChecker's implementation to be tightly coupled with the computation of the evaluation metrics, which impedes obtaining the exact output notifications from the tool.

We refactored the code for each approach , based on consultation with the respective authors, to make it easy to apply to new models and datasets. We removed duplication, made the code more modular and generalizable by decoupling the implementation from specific datasets, models, and evaluation metrics. However, we were unable to refactor SelfChecker's code to obtain the exact output notifications for a given test data, instead of the overall evaluation metrics. We have communicated this issue to the authors.

Table 2: Description of the models.

Dataset	Model	GT		#L	#P	Acc.
		#C	*#I*			
Bank Customers	BM1	856	202	3	2 913	80.91%
	BM2	863	195	3	1 473	81.57%
	BM3	850	208	2	3 001	80.34%
	BM4	846	212	4	24 551	79.96%
	BM5	856	202	3	879	80.91%
	BM6	854	204	3	361	80.72%
	BM7	861	197	3	6 081	81.38%
	BM8	868	190	6	4 641	82.04%
	BM9	857	201	3	627	81.0%
	BM10	520	538	3	627	49.15%
	BM11	520	538	3	627	49.15%
	BM12	869	189	4	1 439	82.14%
German Credit	GC1	67	33	2	1 201	67.0%
	GC2	67	33	2	2 401	67.0%
	GC3	67	33	2	217	67.0%
	GC4	67	33	3	171	67.0%
	GC5	67	33	6	4 257	67.0%
	GC6	67	33	3	2 295	67.0%
	GC7	67	33	3	2 295	67.0%
	GC8	67	33	3	2 295	67.0%
	GC9	67	33	4	2 801	67.0%
House Price	HP1	124	22	3	273	84.93%
	HP2	122	24	3	273	83.56%
	HP3	74	72	3	273	50.68%
	HP4	128	18	4	383	87.67%
Pima Diabetes	PD1	61	16	3	221	79.22%
	PD2	45	32	3	221	58.44%
	PD3	46	31	3	221	59.74%
	PD4	62	15	4	293	80.52%
CIFAR10	ConvNet	8 045	1 955	25	2.9M	80.45%

#C - Num. of Correct; *#I* - Num. of Incorrect;
#L - Num. of Layers; *#P* - Num. of Trainable Parameters;

In order to use Prophecy for the purpose of this study, as explained in Section 2, we used it to build layer-wise pre-conditions to capture the decision logic of the model in terms of features learnt at the layer. Specifically, Prophecy builds a decision-tree estimator in terms of neuron values at every layer, to predict correct and incorrect model behavior. We implemented an efficient algorithm that directly invokes this estimator at inference time for the purpose of runtime detection.

A noteworthy point is that the layers of a model's architecture that are considered by each tool is driven by the design rationale of the approach. DeepInfer is implemented to consider all layers of the model, in order to build pre-conditions at the input layer. SelfChecker, on the other hand, considers only the activation layers to build PDDs based on layer features after each such layer. Prophecy works on the dense and activation layers; it considers only the layers of the classification head for vision models TrustDNN provides the ability to set suitable configuration parameters to enable the layer selection for each tool.

3.3.4 **Metrics.** SelfChecker and DeepInfer use similar metrics but we found discrepancies, as pointed out in the challenges (Section 2.1). Specifically, there is ambiguity in defining the *positive* and *negative* classes when using True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN) metrics. Traditionally, the *positive* class signifies correct predictions, while the *negative* class signifies incorrect ones. However, SelfChecker flips this association, considering incorrect predictions as *positive* and correct ones as *negative*. Although unconventional, we adopt

SelfChecker's approach in our paper as it provides a meaningful evaluation of a technique's performance with respect to its ability to capture misclassifications accurately.

In this study, we propose to use the Matthews Correlation Coefficient (MCC) as our primary metric for comparing the effectiveness of the approaches for several reasons. First, MCC considers all four categories of the confusion matrix, making it immune to class swapping and comprehensively assessing the performance of a tool [11]. This ensures a fair comparison between tools, regardless of how the positive and negative classes are defined. This is clearer when we observe the discrepancy in reported metrics between DeepInfer and SelfChecker. While DeepInfer claims alignment with Self-Checker's metrics, there are apparent inconsistencies in their reported numbers on true positives and false positives, raising concerns about relying solely on traditional metrics. By incorporating MCC, we enhance the reliability and interpretability of our evaluation, mitigating the risk of misinterpretations or discrepancies in reported metrics. Furthermore, MCC considers the distribution of positive and negative elements in the dataset, offering a balanced evaluation of the capability of a tool to classify instances across all classes. This balanced assessment is crucial, especially in detecting misclassified inputs that are intrinsically less frequent.

MCC represents the correlation between the predicted and actual classifications, with values ranging from -1 (total disagreement) to +1 (perfect agreement), where 0 indicates random classification. The MCC calculation is based on the differences between the observed and expected classifications, considering both the overall agreement and the balance between the positive and negative classifications.

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}} \tag{1}$$

To provide a more comprehensive overview of our results, we incorporate metrics such as True Positive Rate (TPR), False Positive Rate (FPR), and F1 Score alongside MCC. While TPR, FPR, and F1 Score may bias the assessment towards the positive class and overlook certain aspects of the confusion matrix, they offer valuable insights when appropriately interpreted. Further, these metrics were used in the previous studies on DeepInfer, SelfChecker, and Prophecy.

- **False Positive Rate (FPR)** quantifies the tendency of a tool to misclassify negative instances as positive with the following equation:

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \tag{2}$$

  FPR evaluation is crucial for tasks prioritizing the reduction of false alarms.

- **True Positive Rate (TPR)**, also known as sensitivity or recall, measures the ability of a tool to identify positive instances correctly and is computed as:

$$\text{TPR (Recall)} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

  TPR evaluation is essential for assessing positive instance detection, especially in scenarios where capturing all positive cases is critical.

- **Precision** assesses the reliability of positive predictions made by the tool, and its usage is particularly relevant in contexts where minimizing false positives is paramount. It is computed with the following:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4}$$

- **F1 Score** provides a balanced assessment by considering the harmonic mean of precision and recall with the following:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

  F1 Score offers a comprehensive view of the overall performance of a tool and is useful in situations characterized by imbalanced positive and negative instances, as it accounts for false positives and negatives.

We would like to draw attention to the fact that the *uncertains* are considered as incorrect in the metric formulae.

4 Evaluation

We aim to address the following **research questions** in this study.

- RQ1: For DeepInfer and SelfChecker, to what extent can the results be reproduced on their own artifacts (datasets and models)?
- RQ2: For the considered approaches, can the results be obtained on artifacts other than their own?
- RQ3: How do the considered approaches compare in terms of effectiveness?
- RQ4: How do the considered approaches compare in terms of time and memory consumption?

4.1 Experimental Design and Setup

We ran two sets of experiments: *replicability analysis* (to answer RQ1 and RQ2) and *comparative analysis* (to answer RQ3 and RQ4).

For RQ1 in the *replicability analysis*, we ran each tool on the exact configuration, models, and datasets the respective replication packages provide.

To address RQ2 and for the *comparative analysis*, we executed each tool using the datasets and models prepared by TrustBench (see Section 3.1). The exact configuration parameters set for each experiment and tool are in our replication package [7].

The experiments were executed on a Debian-based system with the Linux kernel version 5.10.0-16-amd64 running on a remote server with 128 GB of RAM and a 2.1 GHz Intel(R) Xeon(R) Silver 4130 with 48 cores.

Table 3: Replication results for DeepInfer

Model	Ground Truth		DeepInfer				
	#C	*#I*	*#C*	*#I*	*#U*	*#Violation*	*#Satisfaction*
PD1	119	34	108	43	2	192	1 032
PD2	99	54	153	0	0	0	1 224
PD3	98	55	74	79	0	129	1 095
PD4	111	42	37	116	0	132	1 092
HP1	147	145	188	98	6	341	2 579
HP2	147	145	188	98	6	341	2 579
HP3	145	147	292	0	0	0	2 920
HP4	147	145	107	184	1	188	2 732
BM1	1 713	403	616	1 500	0	18 814	40 434
BM2	1 724	392	1 492	624	0	14 855	44 393
BM3	1 707	409	734	1 382	0	7 370	51 878
BM4	1 663	453	1 061	1 055	0	17 486	41 762
BM5	1 734	382	998	1 118	0	11 970	47 278
BM6	1 717	399	609	1 507	0	18 874	40 374
BM7	1 721	395	1 375	741	0	12 762	46 486
BM8	1 732	384	1 089	1 027	0	15 213	44 035
BM9	1 723	393	2 116	0	0	17 090	42 158
BM10	1 092	1 024	1 057	1 059	0	13 972	45 276
BM11	1 092	1 024	863	1 253	0	18 353	40 895
BM12	1 727	389	1 072	1 044	0	17 942	41 306
GC1	198	2	111	89	0	1 044	3 356
GC2	198	2	48	152	0	959	3 441
GC3	198	2	150	50	0	1 417	2 983
GC4	198	2	113	87	0	2 469	1 931
GC5	198	2	63	137	0	1 193	3 207
GC6	198	2	137	63	0	1 328	3 072
GC7	198	2	57	143	0	979	3 421
GC8	198	2	135	65	0	1 507	2 893
GC9	198	2	76	124	0	1 580	2 820

#C - Correct; *#I* - Incorrect; *#U* - Uncertain;

Table 4: Replication results for SelfChecker.

Model	Confusion Matrix				Metrics		
	TP	*FP*	*TN*	*FN*	*TPR*	*FPR*	*F1*
CIFAR10	1 251	239	7 806	704	63.99%	2.97%	72.63%

4.2 Results

RQ.1: For the considered approaches, to what extent can the results be reproduced on their own artifacts?

As Prophecy was not considered before for the problem at hand we do not have a replication package for it. So for this question we only consider SelfChecker and DeepInfer which both come with replication packages. We evaluate DeepInfer on its original datasets (Table 1a) and pre-trained models and SelfChecker with the image dataset (Table 1c) and model.

The paper describing DeepInfer [10] contains two main tables (2 and 3) displaying the results; the corresponding replication package contains two folders (`Table2` and `Table3`) containing the data, models, scripts and other information necessary to reproduce the results in the two tables. Table 3 displays the results of running our implementation of DeepInfer on the data and models from the `Table2` folder in our attempt to replicate the results from Table 2 in the DeepInfer paper [10]. Table 3 displays the numbers that match in green. We summarize our observations below.

*DeepInfer.*We decompose RQ1 into the following two questions: *Are we able to run?* Yes. We were able to run both the original DeepInfer code and our own implementation of it on the datasets provided in the replicated package.

[7] https://zenodo.org/doi/10.5281/zenodo.12803632 in the folder /setup/.trustdnn

Did we get the same results? Only partially. For PD and HP, all the numbers match [10]. The results do not match for the BM and GC models. We looked carefully into the reasons for this and also communicated with the authors of DeepInfer. For the BP models, even the ground truth does not match. We also found that the scripts provided in the replicated package removed one of the branches in the code. In our implementation setup, we run the same code on all the models, avoiding issues such as mentioned above. For GC, we found that the scripts use some pre-computed values that were read from a file. Upon consultation with the authors of DeepInfer, we understand that this was due to variation (randomness) in the inference phase. Although we could not observe such randomness in our experiments, it is possible that randomness can happen due to slight variations of the setup used in experiments performed by different teams.

We further attempted to reproduce the results from Table 3 in [10]. However we noticed some issues that we could not resolve. For instance, Table 3 in [10] contains results for ground truth **ActFP** and **ActTP**. ActTP denotes *if the actual label and predicted label by a model are not equal* and ActFP denotes *if the actual label and predicted label by a model are equal.* Thus, it appears that ActTP is the same as # Incorrect while ActFP is the same as # Correct, yet these numbers seem to be reversed in the Tables in [10]; for instance for PD1, Ground Truth # Correct is 119 in Table 2 [10] while Ground Truth ActTP is also 119 in Table 3 [10]. Further, the replication package for DeepInfer contains two folders containing the same pre-trained models (Table2/Models and Table3/Models) with one exception: by computing the file difference, we found that BM3 in Table3/Models is BM11 in Table2/Models and vice-versa. Another issue is that the scripts under Table3 read some results from files for which we could not find the scripts; upon contacting the authors, they explained how to reproduce the files by changing some hard coded parameters in the scripts under folder Table2. Given all these issues, we concluded that we do not have sufficient information to reliably reproduce the results in Table 3 [10].

SelfChecker. For RQ.1, we again aim to answer the following two questions: *Are we able to run?* Yes. We were able to run both the original SelfChecker code and our own implementation of it on the dataset and model provided in the replicated package. *Did we get the same results?* Yes, we did obtain the same results for the TPR, FPR, and F1 metrics, with minor differences in the numbers. Table 4 displays our results, which reproduce the results from Table 3 in the SelfChecker paper [32].

> ***Observation 1.*** *We found it difficult to reproduce the results. For DeepInfer, we found discrepancies due to the replication of code across multiple scripts, which introduced errors and due to hard coding of some values (an attempt by the authors to reduce variability in the tool's outputs). The unclear definitions for the evaluation metrics used in [10] also lead to different results. For SelfChecker we were able to run the tool on the only model that was made available.*

RQ.2: For the considered approaches, can the results be obtained on artifacts other than their own?

All three tools are designed to process feed-forward neural networks taking in any type of input. We attempt to run them on both image and tabular data. We use the datasets and models curated using TrustBench for this purpose (details in Table 1b, Table 1c and Table 2).

DeepInfer. We were not able to run DeepInfer on image data. The tool throws the following error, `"ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 32)"`. This indicates that the tool is not able to handle the dimensionality and format of image inputs. Consultation with the authors further revealed that the tool, in its current implementation, cannot handle architectures with activation functions in separate individual layers, and the softmax activation function. Therefore, the answer to RQ.2 for DeepInfer is that it cannot be run on image data in its current implementation.

However, we attempted to address this limitation. We extended DeepInfer to enable application to image datasets, by building preconditions at an inner layer (of lower dimensionality) instead of the input layer; we further modified DeepInfer to skip the layers that it can not handle. This extension enables the tool to run on the CIFAR-10 dataset and model. The results are in Table 6 and will be discussed in RQ.3.

SelfChecker. We were not able to run SelfChecker on tabular data. The SelfChecker tool throws a `np.linalg.LinAlgError` in the Kernel density estimation function, when run on the models for tabular data. A similar error regarding SelfChecker was reported in [10] as well.Therefore, the answer to RQ.2 for SelfChecker is that it cannot be run on tabular data in its current implementation.

However, we attempted to address this limitation. In our implementation of SelfChecker, we expanded the input domain to include tabular data by converting it to the expected format (with Pandas). To resolve the error encountered during KDE generation, we implemented regularization in covariance matrix computation. This regularization ensures the matrix remains non-singular and invertible, with an alpha value of 0.1 determining the regularization strength. The existing implementation uses Guassian KDE estimation. We specifically modified the kde estimation to use Regularized KDE with the following parameters `bw_method= scott` and `alpha= 0.01`, when the above error was caught. These changes enabled running the tool on the tabular data models without any errors. The results are in Table 5 and will be discussed in RQ.3.

Prophecy. The tool ran without any issues on both image and tabular data, which suffices to answer RQ.2. The results are in Table 7 and will be discussed in RQ.3.

> ***Observation 2.*** *We found it difficult to produce results for the considered approaches using artifacts other than their own. With their original implementations, DeepInfer cannot be run on image data, and SelfChecker cannot be run on tabular data. By changing the KDE function, we ran SelfChecker on tabular data (and respective models). DeepInfer required a bigger change to apply it to image models. Prophecy worked on both tabular and image data without modification.*

RQ.3: How do the considered approaches compare in terms of effectiveness in determining mis-classifications?

We use the same datasets as for RQ2 and the extended versions of DeepInfer and SelfChecker. We use a common definition to calculate the TP, FP, TN, FN metrics for all three approaches as described in section 3.3.4. Tables 5, 6, and 7 presents the results. [8] We use the MCC metric to compare the effectiveness of the three approaches. Figure 3 summarizes the comparison results.

We present results of only 1 execution, however, we did perform multiple runs for each tool. We found that SelfChecker and DeepInfer produce the same results over multiple runs. Prophecy, on the other hand, displayed variability in the results. We modified the implementation by fixing the "random_state" parameter in the decision-tree estimator, which made the results stable.

DeepInfer performs well across all the models with tabular data. The average MCC is 0.41 indicating good correlation between the predictions and actuals, except for some cases, specifically for BM10 and BM11 models, where the MCC is slightly negative. It is noteworthy that for HP3, BM9 and PD2, the MCC value is 0 (refer Table 6. This is to be interpreted as a random correlation, and occurs when both the TP and FP are zeros, or both TN and FN are zeros. In these specific cases, DeepInfer predicts all instances as correct, leading to zero TPs and FPs.

The performance of SelfChecker and Prophecy across all tabular models is comparable, with average MCC of 0.33 and 0.31, respectively. For GC1, SelfChecker performs poorly (MCC of -0.234), where it has zero true positives, and for BM11, SelfChecker has an MCC of 0 since it predicts all instances as incorrect, leading to zero TN and FN (refer Table 5). Prophecy, on the other hand, does not have zero MCC for any model, indicating that it does not mark all instances as correct or incorrect, for any model. However, it performs poorly on PD2 and PD3 models with negative MCCs (refer Table 7).

For the image case-study, both DeepInfer and SelfChecker perform equally well (MCCs of 0.62 and 0.68 respectively). Prophecy performs poorly on the image model with an MCC of 0.06. This could potentially be attributed to the performance of the decision-tree learning algorithm being adversely impacted by the balance of correctly classified vs mis-classified inputs in the data used in the analyze phase. Given the high accuracy of the ConvNet model on the train data, the efficacy of the mis-classification detection is poor. Enabling Prophecy to use a balanced train dataset in the analysis phase, leads to better performance (MCC 0.497 refer Table 8) corroborates our reasoning.

> ***Observation 3.*** *All tools demonstrate meaningful detections with positive MCC values for most models, with DeepInfer performing the best on average. SelfChecker and Prophecy are comparable in their overall effectiveness, with SelfChecker excelling on the image model. DeepInfer and SelfChecker show high variance in effectiveness across models, while Prophecy has a more balanced performance.*

[8] The full version on [24] has more detailed results tables.

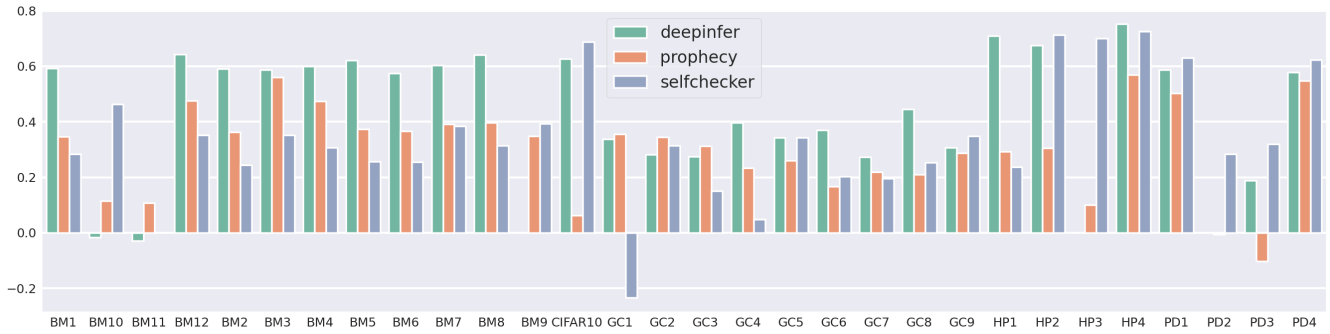

Figure 3: Effectiveness of the approaches in terms of MCC (y-axis) for all models (x-axis).

Table 5: Effectiveness results by model for SelfChecker.

Model	Confusion Matrix				Metrics				
	TP	*FP*	*TN*	*FN*	*TPR*	*FPR*	*Prec.*	*F1*	*MCC*
BM1	538	444	76	0	100%	85.38%	54.79%	70.79%	0.283
BM2	536	457	63	2	99.63%	87.88%	53.98%	70.02%	0.244
BM3	534	397	123	4	99.26%	76.35%	57.36%	72.7%	0.352
BM4	535	424	96	3	99.44%	81.54%	55.79%	71.48%	0.307
BM5	537	454	66	1	99.81%	87.31%	54.19%	70.24%	0.257
BM6	535	450	70	3	99.44%	86.54%	54.31%	70.26%	0.255
BM7	531	372	148	7	98.7%	71.54%	58.8%	73.7%	0.384
BM8	535	420	100	3	99.44%	80.77%	56.02%	71.67%	0.315
BM9	534	373	147	4	99.26%	71.73%	58.88%	73.91%	0.393
BM10	248	28	492	290	46.1%	5.38%	89.86%	60.93%	0.464
BM11	538	520	0	0	100.0%	100.0%	50.85%	67.42%	0.0
BM12	534	397	123	4	99.26%	76.35%	57.36%	72.7%	0.352
CIFAR10	1 244	207	7 838	711	63.63%	2.57%	85.73%	73.05%	0.688
GC1	0	10	57	33	0.0%	14.93%	0.0%	0.0%	-0.234
GC2	19	17	50	14	57.58%	25.37%	52.78%	55.07%	0.315
GC3	22	34	33	11	66.67%	50.75%	39.29%	49.44%	0.151
GC4	17	31	36	16	51.52%	46.27%	35.42%	41.98%	0.049
GC5	20	17	50	13	60.61%	25.37%	54.05%	57.14%	0.343
GC6	12	12	55	21	36.36%	17.91%	50.0%	42.11%	0.203
GC7	13	14	53	20	39.39%	20.9%	48.15%	43.33%	0.196
GC8	13	11	56	20	39.39%	16.42%	54.17%	45.61%	0.253
GC9	17	12	55	16	51.52%	17.91%	58.62%	54.84%	0.348
HP1	72	66	8	0	100.0%	89.19%	52.17%	68.57%	0.237
HP2	60	9	65	12	83.33%	12.16%	86.96%	85.11%	0.713
HP3	58	8	66	14	80.56%	10.81%	87.88%	84.06%	0.701
HP4	62	10	64	10	86.11%	13.51%	86.11%	86.11%	0.726
PD1	26	8	37	6	81.25%	17.78%	76.47%	78.79%	0.63
PD2	11	5	40	21	34.38%	11.11%	68.75%	45.83%	0.283
PD3	29	28	17	3	90.62%	62.22%	50.88%	65.17%	0.319
PD4	24	6	39	8	75.0%	13.33%	80.0%	77.42%	0.623

Table 6: Effectiveness results by model for DeepInfer.

Model	Confusion Matrix				Metrics				
	TP	*FP*	*TN*	*FN*	*TPR*	*FPR*	*Prec.*	*F1*	*MCC*
BM1	597	151	259	51	92.13%	36.83%	79.81%	85.53%	0.592
BM2	246	59	617	136	64.4%	8.73%	80.66%	71.62%	0.59
BM3	560	138	290	70	88.89%	32.24%	80.23%	84.34%	0.587
BM4	422	93	424	119	78.0%	17.99%	81.94%	79.92%	0.6
BM5	427	125	429	77	84.72%	22.56%	77.36%	80.87%	0.621
BM6	611	146	243	58	91.33%	37.53%	80.71%	85.69%	0.575
BM7	288	71	573	126	69.57%	11.02%	80.22%	74.51%	0.603
BM8	428	90	440	100	81.06%	16.98%	82.63%	81.84%	0.641
BM9	0	0	857	201	0.0%	0.0%	0.0%	0.0%	0.0
BM10	260	249	260	289	47.36%	48.92%	51.08%	49.15%	-0.016
BM11	321	292	199	246	56.61%	59.47%	52.37%	54.41%	-0.029
BM12	424	99	445	90	82.49%	18.2%	81.07%	81.77%	0.643
CIFAR10	3 938	1 492	4 107	463	89.48%	26.65%	72.52%	80.11%	0.626
GC1	30	18	37	15	66.67%	32.73%	62.5%	64.52%	0.338
GC2	53	26	14	7	88.33%	65.0%	67.09%	76.26%	0.281
GC3	16	10	51	23	41.03%	16.39%	61.54%	49.23%	0.274
GC4	38	6	29	27	58.46%	17.14%	86.36%	69.72%	0.397
GC5	23	9	44	24	48.94%	16.98%	71.88%	58.23%	0.342
GC6	40	8	27	25	61.54%	22.86%	83.33%	70.8%	0.369
GC7	50	21	17	12	80.65%	55.26%	70.42%	75.19%	0.272
GC8	33	3	34	30	52.38%	8.11%	91.67%	66.67%	0.445
GC9	45	14	22	19	70.31%	38.89%	76.27%	73.17%	0.307
HP1	64	5	60	17	79.01%	7.69%	92.75%	85.33%	0.71
HP2	56	8	66	16	77.78%	10.81%	87.5%	82.35%	0.675
HP3	0	0	74	72	0.0%	0.0%	0.0%	0.0%	0.0
HP4	60	8	68	10	85.71%	10.53%	88.24%	86.96%	0.753
PD1	21	3	40	13	61.76%	6.98%	87.5%	72.41%	0.587
PD2	0	0	45	32	0.0%	0.0%	0.0%	0.0%	0.0
PD3	19	17	27	14	57.58%	38.64%	52.78%	55.07%	0.188
PD4	45	12	17	3	93.75%	41.38%	78.95%	85.71%	0.579

RQ.4: How do the considered approaches compare in terms of time and memory consumption?

Our evaluation of the efficiency of three approaches—DeepInfer, Prophecy, and SelfChecker—revealed negligible differences in total execution duration (both *analyze* and *infer* phases) on the models with tabular data (average 3.5 secs). These models are relatively small in size (refer table 2). However, the times were longer on the bigger ConvNet model for CIFAR10, with SelfChecker consuming the maximum amount of time (32.2 minutes). In terms of the memory consumption, SelfChecker again seems to be very expensive for both types of models (average peak memory usage of 4K Mebibytes for tabular models and 41K Mebibytes for the image model). The memory usage of DeepInfer is lower for tabular models but jumps to 29K Mebibytes for the image model. Prophecy has overall low memory usage for all models. Table 9 presents the details.

Observation 4. *Overall, we can observe that SelfChecker seems to be the most resource intensive, while Prophecy seems to be the least across all models.*

5 Discussion

5.1 Threats to Validity

A tool tasked with assessing the reliability of a pre-trained model needs to execute the model on data "unseen" by the model. This requires knowledge of the exact train/val/test data used to train the model. We are uncertain of the overlap between the train set used to train the model and the data used to train the detector. This can add a threat to the validity of our results. However, we envisage this to be typically the case when handling off-the-shelf models.

Our code modifications, which consist of refactorings, optimizations and extensions, may have introduced errors; we addressed

Table 7: Effectiveness results by model for Prophecy.

Model	Confusion Matrix				Metrics				
	TP	*FP*	*TN*	*FN*	*TPR*	*FPR*	*Prec.*	*F1*	*MCC*
BM1	72	42	784	160	31.03%	5.08%	63.16%	41.62%	0.346
BM2	75	43	788	152	33.04%	5.17%	63.56%	43.48%	0.363
BM3	253	98	597	110	69.7%	14.1%	72.08%	70.87%	0.561
BM4	151	60	695	152	49.83%	7.95%	71.56%	58.75%	0.474
BM5	81	41	775	161	33.47%	5.02%	66.39%	44.51%	0.374
BM6	85	52	769	152	35.86%	6.33%	62.04%	45.45%	0.367
BM7	88	47	773	150	36.97%	5.73%	65.19%	47.18%	0.391
BM8	94	67	774	123	43.32%	7.97%	58.39%	49.74%	0.397
BM9	78	55	779	146	34.82%	6.59%	58.65%	43.7%	0.348
BM10	221	454	299	84	72.46%	60.29%	32.74%	45.1%	0.115
BM11	229	449	291	89	72.01%	60.68%	33.78%	45.98%	0.108
BM12	130	65	739	124	51.18%	8.08%	66.67%	57.91%	0.475
CIFAR10	91	186	7 954	1 769	4.89%	2.29%	32.85%	8.52%	0.062
GC1	35	22	32	11	76.09%	40.74%	61.4%	67.96%	0.356
GC2	27	22	40	11	71.05%	35.48%	55.1%	62.07%	0.345
GC3	23	14	44	19	54.76%	24.14%	62.16%	58.23%	0.313
GC4	14	12	53	21	40.0%	18.46%	53.85%	45.9%	0.234
GC5	17	18	50	15	53.12%	26.47%	48.57%	50.75%	0.261
GC6	10	12	57	21	32.26%	17.39%	45.45%	37.74%	0.166
GC7	13	12	54	21	38.24%	18.18%	52.0%	44.07%	0.219
GC8	9	6	58	27	25.0%	9.38%	60.0%	35.29%	0.21
GC9	20	16	47	17	54.05%	25.4%	55.56%	54.79%	0.288
HP1	6	6	118	16	27.27%	4.84%	50.0%	35.29%	0.292
HP2	7	6	115	18	28.0%	4.96%	53.85%	36.84%	0.305
HP3	19	62	55	10	65.52%	52.99%	23.46%	34.55%	0.101
HP4	16	7	112	11	59.26%	5.88%	69.57%	64.0%	0.569
PD	11	2	50	14	44.0%	3.85%	84.62%	57.89%	0.502
PD2	5	9	40	23	17.86%	18.37%	35.71%	23.81%	-0.006
PD3	2	8	44	23	8.0%	15.38%	20.0%	11.43%	-0.103
PD4	15	4	47	11	57.69%	7.84%	78.95%	66.67%	0.547

Table 8: Effectiveness of Prophecy on CIFAR10 with balanced train data.

Model	Confusion Matrix				Metrics				
	TP	*FP*	*TN*	*FN*	*TPR*	*FPR*	*Prec.*	*F1*	*MCC*
CIFAR10	1 511	1 423	6 534	532	73.96%	17.88%	51.5%	60.72%	0.497

Table 9: Time (average duration in seconds) and Memory (peak memory usage in Mebibytes) efficiency for each approach by dataset.

Tool	Phase	Dataset	Duration	Memory
DeepInfer	analyze	BM	3.45	687.58
	analyze	CIFAR10	55.72	29 150.96
	analyze	GC	4.89	661.32
	analyze	HP	2.82	643.71
	analyze	PD	2.67	659.31
	infer	BM	2.63	646.03
	infer	CIFAR10	29.01	29 146.96
	infer	GC	2.64	642.5
	infer	HP	2.62	646.69
	infer	PD	2.67	648.37
Prophecy	analyze	BM	3.94	730.33
	analyze	CIFAR10	118.42	4 177.04
	analyze	GC	3.33	711.58
	analyze	HP	3.22	709.97
	analyze	PD	3.27	710.97
	infer	BM	3.49	696.4
	infer	CIFAR10	25.97	2 012.66
	infer	GC	3.04	696.16
	infer	HP	3.07	699.33
	infer	PD	3.02	699.5
SelfChecker	analyze	BM	9.05	4 063.45
	analyze	CIFAR10	1 930.52	40 871.41
	analyze	GC	3.51	3 959.27
	analyze	HP	3.47	4 069.1
	analyze	PD	3.52	3 902.47
	infer	BM	4.44	3 773.3
	infer	CIFAR10	1 352.59	7 363.5
	infer	GC	3.22	3 796.63
	infer	HP	3.17	3 833.67
	infer	PD	3.17	3 758.48

by being able to reproduce the results for DeepInfer (partially) and SelfChecker (fully). Although we performed preliminary experiments to analyze variability of results over multiple executions, this may not be sufficient to make reliable conclusions. Further, the modification made to Prophecy's code to stabilize its results introduces threat to validity. The number and type of datasets and models used threatens the external validity of our results regarding the tools.

5.2 Lessons Learned

Our study emphasizes again the need for open science. Beyond the application considered in this study, in order to replicate and evaluate tools in general, we would like to emphasize on the following key characteristics.

- *Transparency:* Tool repositories should store precise artifacts that act as inputs (datasets, models so on), store experimental results, add proper documentation explaining usage and precise configuration parameters.
- *Generalizability and Extensibility:* In order to apply the tool to various benchmarks, the code should avoid hard-coding parameters. The code should be modular and not tightly coupled.The repository structure should also be modular, avoid duplication of scripts, and apply same code consistently across all benchmarks.
- *Handling variability in results:* Although we did not notice variability in our study, the inherent randomness of machine learning models necessitates a large number of trials to ensure reliability of the results. We also recommend using a docker technology when building replication package.

We hope the *TrustBench* and the *TrustDNN* framework aid in addressing some of the lessons learned for future tools.

6 Conclusion

In this paper, we evaluated recent approaches that have been proposed for evaluating the reliability of DNNs. We found that it is difficult to run and reproduce the results for these approaches on their replication packages, and it is also difficult to run them on artifacts other than their own. Further, it is difficult to compare the effectiveness of the approaches, as they use different evaluation metrics. Our results indicate that more effort is needed in our research community to obtain sound techniques for evaluating the reliability of DNNs. To this end, we contribute an evaluation framework that we make available as open source. The framework enables experimentation with different approaches for evaluating reliability of DNNs, allowing comparison using clear metrics. We hope that the community can build on the framework that we provide and continue research on this challenging topic.

Acknowledgments

This work was partially supported by FCT - Fundação para a Ciência e Tecnologia, I.P. by project reference PRT/BD/152197/2021 and DOI identifier https://doi.org/10.54499/PRT/BD/152197/2021. We would like to thank Ahmed Shibbir for his insightful feedback on the replication package for DeepInfer.

References

[1] [n. d.]. Keras documentation: CIFAR10 small images classification dataset. https://keras.io/api/datasets/cifar10/ Accessed on July 24, 2024.

[2] 2016. Pima Indians Diabetes Dataset. https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database Accessed on July 24, 2024.

[3] 2018. Bank Marketing Dataset. https://www.kaggle.com/datasets/sharanmk/bank-marketing-term-deposit Accessed on July 24, 2024.

[4] 2019. German Credit Risk Classification with Keras. https://www.kaggle.com/code/twunderbar/german-credit-risk-classification-with-keras/data Accessed on July 24, 2024.

[5] 2019. House Price Prediction Dataset. https://www.kaggle.com/datasets/moewie94/housepricedata Accessed on July 24, 2024.

[6] 2019. Kaggle Notebook for preprocessing German Credit dataset. https://www.kaggle.com/code/twunderbar/german-credit-risk-classification-with-keras/notebook Accessed on July 24, 2024.

[7] 2019. Kaggle Notebook for preprocessing House Price dataset. https://www.kaggle.com/code/moewie94/neural-network-with-keras/notebook Accessed on July 24, 2024.

[8] 2021. Kaggle Notebook for preprocessing Bank Marketing dataset. https://www.kaggle.com/code/rxsraghavagrawal/build-your-first-ann-using-keras-on-bank-customer Accessed on July 24, 2024.

[9] Shibbir Ahmed, Hongyang Gao, and Hridesh Rajan. 2023. Repository of DeepInfer. https://github.com/shibbirtanvin/DeepInfer

[10] Shibbir Ahmed, Hongyang Gao, and Hridesh Rajan. 2024. Inferring Data Preconditions from Deep Learning Models for Trustworthy Prediction in Deployment. arXiv:2401.14628 [cs.SE]

[11] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics* 21 (2020). https://api.semanticscholar.org/CorpusID:209528322

[12] Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. 2012. Deep neural networks segment neuronal membranes in electron microscopy images. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 2843–2851.

[13] Charles Corbière, Nicolas THOME, Avner Bar-Hen, Matthieu Cord, and Patrick Pérez. 2019. Addressing Failure Prediction by Learning Model Confidence. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/757f843a169cc678064d9530d12a1881-Paper.pdf

[14] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Alexandra Meliou. 2020. Data Invariants: On Trust in Data-Driven Systems. *CoRR* abs/2003.01289 (2020). arXiv:2003.01289 https://arxiv.org/abs/2003.01289

[15] Divya Gopinath, Hayes Converse, Corina S. Păsăreanu, and Ankur Taly. 2019. Property Inference for Deep Neural Networks. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), 797–809. https://api.semanticscholar.org/CorpusID:202577825

[16] J. Gray. 1991. *The Benchmark Handbook: For Database and Transaction Processing Systems*. M. Kaufmann Publishers. https://books.google.com/books?id=UKQXAAAAIAAJ

[17] Dan Hendrycks and Kevin Gimpel. 2016. A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks. *CoRR* abs/1610.02136 (2016). arXiv:1610.02136 http://arxiv.org/abs/1610.02136

[18] Joel Janai, Fatma Güney, Aseem Behl, and Andreas Geiger. 2020. Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art. *Found. Trends. Comput. Graph. Vis.* 12, 1–3 (jul 2020), 1–308. https://doi.org/10.1561/0600000079

[19] Heinrich Jiang, Been Kim, Melody Y. Guan, and Maya Gupta. 2018. To Trust Or Not To Trust A Classifier. arXiv:1805.11783 [stat.ML]

[20] Rasmus Kær Jørgensen and Christian Igel. 2021. Machine learning for financial transaction classification across companies using character-level word embeddings of text fields. *Intell. Syst. Account. Finance Manag.* 28, 3 (2021), 159–172. https://doi.org/10.1002/ISAF.1500

[21] Ismet Burak Kadron, Divya Gopinath, Corina S. Pasareanu, and Huafeng Yu. 2021. Case Study: Analysis of Autonomous Center Line Tracking Neural Networks. In *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13124)*, Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina (Eds.). Springer, 104–121. https://doi.org/10.1007/978-3-030-95561-8_7

[22] Edward Kim, Divya Gopinath, Corina S. Pasareanu, and Sanjit A. Seshia. 2020. A Programmatic and Semantic Approach to Explaining and Debugging Neural Network Based Object Detectors. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 11125–11134. https://doi.org/10.1109/CVPR42600.2020.01114

[23] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. https://api.semanticscholar.org/CorpusID:18268744

[24] Eduard Pinconschi, Divya Gopinath, Rui Abreu, and Corina S. Pasareanu. 2024. Evaluating Deep Neural Networks in Deployment (A Comparative and Replicability Study). arXiv:2407.08730 [cs.NE] https://arxiv.org/abs/2407.08730

[25] Don Roberts and Ralph E. Johnson. 2004. Evolving Frameworks A Pattern Language for Developing Object-Oriented Frameworks. https://api.semanticscholar.org/CorpusID:10900135

[26] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour prediction for autonomous driving systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 359–371. https://doi.org/10.1145/3377811.3380353

[27] Muhammad Usman, Divya Gopinath, Youcheng Sun, Yannic Noller, and Corina S. Pasareanu. 2021. NNrepair: Constraint-Based Repair of Neural Network Classifiers. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 3–25. https://doi.org/10.1007/978-3-030-81685-8_1

[28] Muhammad Usman, Divya Gopinath, Youcheng Sun, and Corina S. Pasareanu. 2022. Rule-Based Runtime Mitigation Against Poison Attacks on Neural Networks. In *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13498)*, Thao Dang and Volker Stolz (Eds.). Springer, 67–84. https://doi.org/10.1007/978-3-031-17196-3_4

[29] Muhammad Usman, Youcheng Sun, Divya Gopinath, and Corina S. Pasareanu. 2023. Rule-Based Testing of Neural Networks. In *Proceedings of the 1st International Workshop on Dependability and Trustworthiness of Safety-Critical Systems with Machine Learned Components, SE4SafeML 2023, San Francisco, CA, USA, 4 December 2023*, Marsha Chechik, Sebastian G. Elbaum, Boyue Caroline Hu, Lina Marsso, and Meriel von Stein (Eds.). ACM, 1–5. https://doi.org/10.1145/3617574.3622747

[30] Vishal Thanvantri Vasudevan, Abhinav Sethy, and Alireza Roshan Ghias. 2019. Towards Better Confidence Estimation for Neural Models. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 7335–7339. https://doi.org/10.1109/ICASSP.2019.8683359

[31] Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Dissector: input validation for deep learning applications by crossing-layer dissection. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 727–738. https://doi.org/10.1145/3377811.3380379

[32] Yan Xiao, Ivan Beschastnikh, David S. Rosenblum, Changsheng Sun, Sebastian G. Elbaum, Yun Lin, and Jin Song Dong. 2021. Self-Checking Deep Neural Networks in Deployment. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 372–384. https://api.semanticscholar.org/CorpusID:230125000

[33] Yan Xiao, Ivan Beschastnikhy, David S. Rosenblumz, Changsheng Sun, Sebastian Elbaum, Yun Lin and Jin Song Dong. 2021. Repository of SelfChecker. https://github.com/self-checker/SelfChecker