QuantifyML: How good is my machine learning model?

Anonymous Author(s)

ABSTRACT

This paper presents an approach, *QuantifyML*, which employs model counting to assess the *learnability* and *robustness* of machine learning models. Typically the efficacy of machine learning models is determined by computing their accuracy statistically on test data sets. However, this may be misleading, if the test data is not representative of the problem that is being studied. Further, two different models may have the same accuracy on a given data set, measured statistically, but may be very different in their behavior on unseen data. Also, models with high accuracy could have poor adversarial robustness. In *QuantifyML*, our goal is to *precisely* quantify the extent to which machine learning models have learned and generalized from the given data. In *QuantifyML*, a trained model is translated into a C program, which is fed to the CBMC model checking tool to produce a formula in Conjunctive Normal Form (CNF), which in turn is analyzed with state-of-the-art model counters to efficiently obtain precise counts w.r.t different outputs. QuantifyML enables i) evaluating the learnability of models by comparing the counts for the outputs to ground truth, expressed as logical predicates (if available), ii) comparing the performance of different models that may be built with different machine learning algorithms (e.g., decision-trees vs. neural networks), and iii) quantifying the robustness of trained models around given inputs. Our evaluation demonstrates these applications of QuantifyML on decision trees and neural networks trained to learn relational properties of graphs, for which we know the ground truth, and to perform image classification, for which we do not have the ground truth, but we can quantify local robustness.

ACM Reference Format:
Anonymous Author(s). 2021. QuantifyML: How good is my machine learning model?. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent years have seen a surge in the use of machine learning algorithms in a variety of applications to analyze and learn from huge amounts of data. For instance, decision-trees are a popular class of supervised learning techniques that can learn easily interpretable rules from data. They have found success in areas such as medical diagnosis and credit scoring [1, 2]. Another example is Deep Neural Networks (DNN), which have gained immense popularity in fields such as banking, health-care, image and speech recognition, as well as perception in self-driving cars [3, 4].

Machine learning models are typically evaluated statistically, by computing their *accuracy* on test sets, to determine how well a model learned and generalized from the train sets. However, this is an imperfect measure, as the train and test sets may not cover well the inputs space. Furthermore, it is often not clear which learning algorithm or trained model is better suited for a particular problem (e.g., neural networks vs. decision trees), and simply comparing the accuracy of different models may lead to misleading results. It is also the case that machine learning models that have high accuracy on test sets may have poor accuracy on *adversarial inputs*, which are inputs that are very similar to validly classified inputs but fool the network into mis-classification [5–7].

In this paper, we present an approach, *QuantifyML*, that aims to *precisely quantify* the learnability and robustness of machine learning models. A given model is translated into a C program, enabling the application of the CBMC tool [8] to translate it to the Conjunctive Normal Form (CNF), which in turn enables the application of state-of-the-art approximate and exact model counters [9–11] to obtain precise counts w.r.t different output classes. QuantifyML enables i) evaluating the learnability of models by comparing the counts for the outputs to ground truth (expressed as logical predicates, if available), ii) comparing the performance of different models that may be built with different machine learning algorithms (e.g., decision-trees vs. neural networks), and iii) quantifying the robustness of neural network models. Our evaluation demonstrates these applications of QuantifyML on decision-tree and neural network classifiers for the problems of learning relational properties of graphs and image classification.

We derive inspiration from the recent paper [12] which presents the *Model Counting meets Machine Learning* approach to evaluate the learnability of binary decision trees. QuantifyML generalizes MCML by providing a generic procedure that can handle more realistic multi-class problems, such as decision trees with non-binary inputs and with more than two output decisions, and also neural networks. Other learning algorithms can be accommodated provided that the models can be translated into C programs. As a consequence, QuantifyML applications beyond MCML include comparison of the performance of different models, built with different learning algorithms, and quantification of robustness in image classifiers, as we demonstrate in this paper.

The key contribution of *QuantifyML* is enabling precise quantification of the performance of any machine learning model in a bounded input space. This has several benefits as enlisted below.

(1) **Evaluation of the true performance of models.** *QuantifyML* enables evaluating the true accuracy of a model by comparing the actual outputs with ideal values (ground truth). We present a study quantifying the performance of binary classifiers that learn relational properties of graphs. Such classifiers can be used in software engineering tasks to enable efficient run-time monitoring and error-recovery [13, 14].

ISSTA 2021, 12-16 July, 2021, Aarhus, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

QuantifyML leverages the available ground truth knowledge to obtain an accurate determination of the performance of the classifiers, thereby increasing confidence in their behavior.

(2) **Comparing the performance of different models and learning algorithms.** An interesting application of this generic technique that enables a standard method of quantifying performance is that it makes it possible to compare different models and learning algorithms. For a given application such as learning relational properties of graphs, *QuantifyML* could be used to compare the performance of different models; learnt using the same or different learning algorithms, and different data sets. Choosing the right learning algorithm for the given problem is a challenge and is mostly determined heuristically based on the data sets available, taking into account dimensionality of data, expected robustness of the predictions so on. *QuantifyML* has the potential to help understand the efficacy of different algorithms for a given problem. To this end, we present a study that compares the performance of decision-tree models and neural network models for the application of learning graph properties.

(3) **Quantifying robustness in image classifiers.** Machine learning models, specifically those used in image classification, have been shown to be vulnerable to adversarial perturbations, i.e. small input perturbations designed to fool the network. There has been extensive work [15, 16] in evaluating the robustness of image classification models by searching for adversarial inputs within a defined neighborhood of correctly classified images where the output label of the classifier is expected to be the same. When such an adversary is found, existing techniques declare the model to not be robust and do not provide any further information about how vulnerable the model is. We present an application of *QuantifyML*, where we cast this problem to boolean satisfiability and leverage model counting to obtain a precise quantification of the robustness for image classification models.

This paper is organized as follows. Section 2 gives the essential background and briefly explains decision trees, neural networks, bounded model checking, and model counting. Section 3 gives detailed information about the QuantifyML framework. Results of experimental evaluation are discussed in section 4 and related work is discussed in section 5. Finally, we discuss the limitations, future work and conclusion in section 6.

2 BACKGROUND

2.1 Decision Trees

Decision tree learning [17] is a supervised learning technique for extracting rules that act as classifiers. Given a set of data labeled to respective classes, decision tree learning aims to discover rules in terms of the attributes of the data to discriminate one label from the other. It builds a tree such that each path of the tree encodes a rule as a conjunction of predicates on the data attributes. Each rule attempts to cluster or group inputs that belong to a certain label. The technique performs a greedy search over a space of functions, being guided by the distribution of data and its labels.

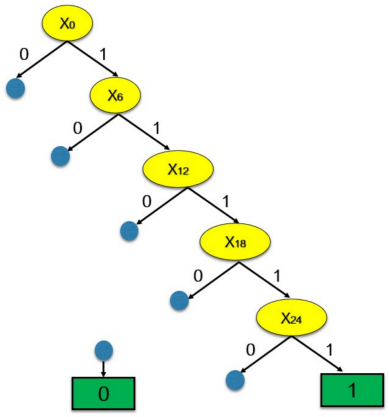

Figure 1: Decision Tree for Reflexive Property

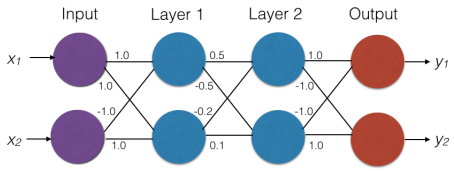

Figure 2: Neural Network

The way this greedy search works is to first identify a predicate that classifies most of the data correctly, and then to iteratively identify additional predicates as required to classify the remaining data. Many decision-tree learning algorithms (such as ID3 [18]) use the concept of information entropy to automatically choose the most advantageous splitting predicate. Figure 1 shows a decision tree for the reflexive relational property that we analyze in our experiments (for a graph with 5 nodes).

2.2 Neural Networks

Neural networks [19] are machine learning algorithms that can be trained to perform different tasks such as classification and regression. Neural networks consist of multiple layers, starting from the *input* layer, followed by one or more *hidden* layers (such as convolutional, dense, activation, and pooling), and a final *decision* layer. Each layer consists of a number of computational units, called *neurons*. Each neuron applies an activation function on a weighted sum of its inputs; $N(X) = \sigma(\sum_i w_i \cdot N_i(X) + b)$ where N_i denotes the value of the i^{th} neuron in the previous layer of the network and the coefficients w_i and the constant b are referred to as *weights* and *bias*, respectively; σ represents the activation function. For instance, the ReLU (rectified linear unit) activation function returns its input as is if it is positive, and returns 0 otherwise, i.e., $\sigma(X) = max(0, X)$. The final decision layer (also known as *logits*) typically uses a specialized function (e.g., max or *softmax*) to determine the decision or the output of the network. Figure 2 shows a neural network with one input layer, two hidden layers and one output layer.

2.3 Bounded Model Checking for C programs

Bounded model checking [20] is a popular technique for verifying safety properties of software systems. Given a bound on the input domain and a bound on the length of executions, a boolean formula is generated that is satisfiable if there exists an error trace or counter-example to the given property. The formula is checked using off-the-shelf decision procedures. CBMC [21] is a tool that performs analysis of programs written in a high-level language such as C, C++ or Java by applying bounded model checking. The program is first converted into a control flow graph (CGF) representation and formulas are built for the paths in the CFG leading to assertions. The model checking problem is reduced to determining the validity of a set of bit-vector equations, which are then flatted out to conjunctive normal form (CNF) and checked for satisfiability.

In this work, we leverage CBMC to build the CNF formulas corresponding to the paths in the C program representation of a machine learning model. We then pass on the formulas corresponding to the respective output classes to a model counting tool in order to quantify the number of solutions.

2.4 Model Counters

We used two state-of-the-art model counters i.e., projMC[1] and ApproxMC[2]. **projMC**[22] is a state-of-the-art exact model counter that outputs the exact number of solutions for a set of constraints and uses a recursive algorithm specifically defined for deterministic disjunctive form. Pairwise variable independent partitions of the original formula are created using disjunctive decomposition. The number of solutions in each partition is then calculated and combined as a total model count of the formula. In case, one of the partition reports a contradiction, the formula is reported to be an UNSAT formula with a model count of 0.

ApproxMC [23] is a state-of-the-art approximate model counter that give counts within a given confidence level and tolerance score. ApproxMC iteratively partitions the solution space into smaller regions such that each region contains the same number of solutions. It then counts the number of solutions in one region and then uses estimation techniques to compute the total number of solutions across all regions. It uses universal hashing technique and a specialized SAT solver (CryptoMiniSat [24]). With the passage of time, ApproxMC was optimized for performance and number of SAT calls was reduced from O(n) to O(log(n)). This was achieved using dependency information between the SAT calls. Recent studies have shown ApproxMC to be efficient and quite close to the exact model counts.

2.4.1 Projected model counting. Many tools, including CBMC, translate a boolean formula to CNF by introducing auxiliary variables. These variables do not affect the satisfiability of the boolean formula but do affect the model counts. In such scenarios, projected model counting [25] needs to be used.

Consider the set M consisting of all variables in a boolean formula and N be a subset of variables in the formula. The solutions in which the value of at least one variable in N is different is considered a unique solution in the *projected* model counting problem. The variables in N are known as primary variables and the rest of the variables are known as auxiliary variables.

Please refer to [26] for a detailed discussion on model counting, projected model counting and model counters. In our work we use projected model counting, where the inputs to the model are considered as primary variables.

[1] http://www.cril.univ-artois.fr/kc/projmc.html

[2] https://github.com/meelgroup/ApproxMC (ApproxMCv3, Git commit ID 23d4439) with the default seed (i.e., 1)

3 APPROACH

Figure 3 shows the overall framework of QuantifyML. Users can input the trained machine learning model (Decision Tree or Neural Network) into QuantifyML. The *ML to C* translator will generate the C programming language representation of the machine learning model. In applications where the ground-truth or an oracle of correctness is available, it is also encoded as a function in C. The C file also contains assertions encoding various checks involving the outputs of the model and of the ground-truth predicates. The final C program is then converted into CNF form using the CBMC tool. The obtained CNF files are then fed to the model counters, allowing us to quantify the quality of the models.

3.1 Translation of Decision Trees to C Programs

We illustrate the translation via an example. Figure 4 shows the C program representation of the decision tree shown in Figure 1. The decision tree is trained on the reflexive property of graphs, i.e., all nodes in the graph have a self loop (all diagonal elements in the adjacency matrix should be 1). The input is an array of size 25 (since the tree is trained on graphs with 5 nodes - represented using an adjacency matrix of size 25). Function *decisiontree* computes and returns the classification label for the given input. The function returns label 0 if any of the diagonal elements is 0 otherwise it returns label 1. The *ML to C* translator currently supports decision trees trained using the Scikit-Learn [27] machine learning library. We plan to add support for other machine learning libraries in the future.

3.2 Translation of Neural Networks to C Programs

QuantifyML can handle feed-forward neural networks with dense, convolutional, and pooling layers, with ReLU activations and Softmax functions. Figure 5 shows the C program representation of the example neural network shown in Figure 2. Function *neural-network* computes and returns the classification label for the given input. There are two inputs to the function i.e., x1 and x2. Line 2 declares an array, *layer1*, of dimension 2 (layer 1 has 2 nodes) that will store the values of the nodes at layer 1. Line 3 and 4 computes the value of nodes at layer 1 by multiplying the inputs with the weights on the respective edges. Line 6 declares an array, *layer2*, of dimension 2 (layer 2 has 2 nodes) that will store the values of the nodes at layer 2. Line 7 and 8 computes the value of nodes at layer 2 by multiplying the values of nodes at layer 1 with the weights on the respective edges. Line 10 declares an array, *output*, of dimension 2 (output layer has 2 nodes) that will store the values of the nodes at output layer. Line 11 and 12 computes the value of nodes at output layer by multiplying the values of nodes at layer 2 with the weights on the respective edges. Line 14-18 returns label 0 if the value of

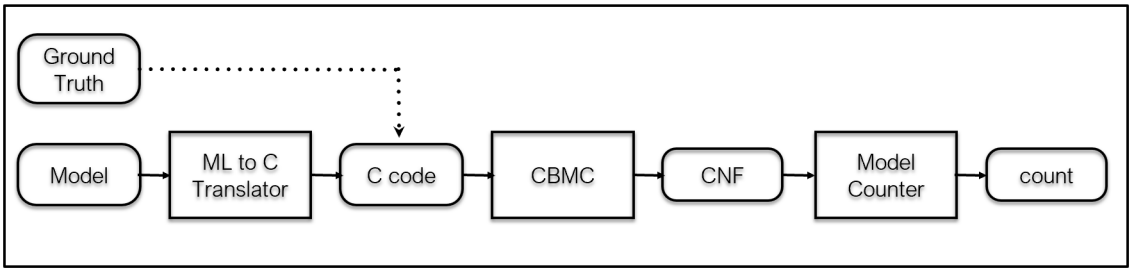

Figure 3: QuantifyML Framework

```
int decisiontree(){
  float x[25];
  if (x[0] == 0){
      return (0);
    }
  else if (x[6] == 0){
      return (0);
    }
  else if (x[12]==0){
      return (0);
    }
  else if (x[18]==0){
      return (0);
    }
  else if (x[24]==0){
      return (0);
    }
    else{
      return (1);
    }
}
```

Figure 4: C code for the decision tree from Figure 1

```
int neuralnetwork(float x1,float x2){
   float layer1[2];
   layer1[0]=(+1.0)*x1+(-1.0)*x2;
   layer1[1]=(+1.0)*x1+(-1.0)*x2;

   float layer2[2];
   layer2[0]=(+0.5)*layer1[0]+(-0.20)*layer1[1];
   layer2[1]=(-0.5)*layer1[0]+(+0.10)*layer1[1];

   float output[2];
   output[0]=(+1.0)*layer2[0]+(-1.0)*layer2[1];
   output[1]=(-1.0)*layer2[0]+(+1.0)*layer2[1];

   if(output[0]>output[1]){
     return 0;
   }
   else{
     return 1;
   }
```

Figure 5: C code for the neural network from Figure 2

first node at output layer (y1) is greater than the value of the second node at the output layer (y2), otherwise it returns 1. QuantifyML *ML to C* translator currently supports neural networks trained in Keras [28] machine learning library. We plan to add support for other machine learning libraries in the future.

3.3 Quantifying the learnability of machine learning models

Given the C program representations of the model and the oracle encoding the ground-truth, *QuantifyML* employs the CBMC tool to generate formulas encoded in the CNF form. The CNF files are then fed to model counters to obtain counts which are in turn used to calculate the following metrics, to quantify the learnability of the models, given finite bounds on the input space.

For each output label l of the classifier, *QuantifyML* computes the following: i) *True Positives (TP)* denotes the portion of the inputs for which the ground truth is the label l and the classifier model correctly predicts the label to be l, ii) *False Positives (FP)* denotes the portion of the inputs for which the ground truth is not the label l and the classifier model incorrectly predicts the label to be l, iii) *True Negatives (TN)* denotes the portion of the inputs for which the ground truth is not the label l and the classifier model correctly predicts a different label than l, and iv) *False Negatives (FN)* denotes the portion of the inputs for which the ground truth is the label l and the classifier model incorrectly predicts a different label than l. It then uses these counts to assess the quality of the model using standard measures such as *Accuracy*, *Precision*, *Recall* and *F1-score* for the model.

- *Accuracy*= $\frac{TP+TN}{TP+FP+TN+FN}$;
- *Precision*= $\frac{TP}{TP+FP}$;
- *Recall*= $\frac{TP}{TP+FN}$;
- *F1-score*= $\frac{2*Precision*Recall}{Precision+Recall}$.

In order to compute these measures for each label l, a predicate function in C is generated ($\phi_l(x)$) which returns 1 if the output of the model is l for a given input x and returns 0 otherwise. Similarly, a predicate function ($\psi_l(x)$) is generated which returns 1 if the ground-truth for the given input x is l and returns 0 otherwise. Given these predicates, four CNF files are generated for each label l, which encode the following formulas corresponding to the counts. N is the scope or bound on the input domain, CNF represents a function that translates C program to formulas in the CNF form, and MC represents a function that uses the projected model counter to return the number of solutions projected to the input variables.

- *True Positives (TP):* $MC(\ CNF(\ \psi_l(x)\ \wedge\ \phi_l(x)\ ),N)$
- *False Positives (FP):* $MC(\ CNF(\ \neg\psi_l(x)\ \wedge\ \phi_l(x)\ ),N)$
- *True Negatives (TN):* $MC(\ CNF(\ \neg\psi_l(x)\ \wedge\ \neg\phi_l(x)\ ),N)$
- *False Negatives (FN):* $MC(\ CNF(\ \psi_l(x)\ \wedge\ \neg\phi_l(x)\ ),N)$

CBMC generates the above mentioned set of CNF files which are then passed to an off-the-shelf model counter and final counts are obtained. Performance metrics as explained above are then calculated. We use two model counters, ApproxMC and projMC, for the experiments in this paper but any model counter that supports projected model counting can also be used. CBMC translates the C program into CNF and introduces auxiliary variables. Therefore, only model counters supporting projected model counting should be used.

3.4 Quantifying Local Robustness

As mentioned, *QuantifyML* is more general than *MCML*, which is applicable only to decision trees where both inputs and decisions are binary. *QuantifyML* can be applied to more challenging models, which can not be described with binary decision trees. The challenge with the analysis of more realistic models is that we typically do not have the ground truth. Image classification is such a problem.

Consider that given an image which contains an object of interest such as a digit, the goal is to classify it with a label that identifies the digit. It is not feasible to define a specification that can automatically generate the correct label or the ground truth for any arbitrary image. Machine learning models are typically trained using a data set of images that are manually labelled. However, images that are similar to, or are in close proximity (in terms of distance in the input space) to an image with a known label can be expected to have the same label. This property is called *robustness* in the literature. For instance, given an image of a certain digit, all images that can be generated by applying perturbations that are imperceptible to the human eye should represent the same digit. Therefore, we know the ground truths for the inputs in the neighborhood of user labelled inputs.

Evaluating the robustness of image classification models is an active area of research [29–31]. Current techniques typically search for the existence of an adversarial input (x') within an ϵ ball surrounding a labelled input (x); e.g., $||x - x'||_\infty \leq \epsilon$ (here the distance is in terms of the L_∞ metric) such that the output of the model on x and x' is different. When no such input exists, the model is declared robust, however, in the presence of an adversarial input there is no further information available. We propose to use *QuantifyML* to *quantify* robustness of machine learning models, where instead of using a predicate encoding the ground truth, we encode the local robustness requirement that the model should give the same output within the region defined by $||x - x'||_\infty \leq \epsilon$.

In order to quantify local robustness around a concrete n-dimensional input $x = (x_0, x_1, ..x_n)$, we first define an input region R_ϵ by constraining the inputs across each dimension to be within $[x_i - \epsilon, x_i + \epsilon]$ in the translated C program. We then define $Robustness_\epsilon$ as $\frac{MC(CNF(\phi_l(x)), R_\epsilon)}{|R_\epsilon|}$, where $\phi_l(x)$ is defined as before as a predicate which returns 1 if the output of the model is l and 0 otherwise, R_ϵ defines the scope for the check, and $|R_\epsilon|$ quantifies its size. Intuitively, $Robustness_\epsilon$ quantifies the portion of the input on which the model is robust, within the small region described by R_ϵ.

4 EVALUATION

We seek to address the following research questions in our evaluation.

- **RQ 1:** How does *QuantifyML* perform when applied to the problem of evaluating the true performance of a model in comparison with the ground truth?
- **RQ 2:** Does *QuantifyML* enable a comparison of different models, that are built using different learning techniques (decision-tree learning vs. neural networks)?
- **RQ 3:** How does *QuantifyML* perform when applied to the problem of quantifying adversarial robustness for image classification models?

All experiments were performed on Windows 10 with an Intel Core-i7 8750H CPU (2.20 GHz) and 16GB RAM. The detailed results along with the artifacts of the analysis are available at the project's GitHub repository[3].

4.1 RQ 1: Evaluating the true performance of the models.

In the first set of experiments we seek to evaluate the performance of trained models against ground truth, encoded as predicates (as described in Section 3). For this set of experiments we use the same set up as in [12], allowing us to also compare the performance of MCML with QuantifyML. We recap that setup here.

4.1.1 Data sets. We consider the problem of learning relational properties of graphs. Alloy [32] was used to create datasets for 11 relational properties of graphs including antisymmetric, connex, equivalence, irreflexive, non-strict order, partial order, pre-order, reflexive, strictorder, total order and transitive. Alloy generated datasets containing positive solutions i.e., graphs which satisfy a given relational property and negative solutions i.e., graphs which violate the given property. The number of negative solutions is much larger than the number of positive solutions. Table 7 in supplementary material tabulates the total number of positive and negative solutions for each property. It is infeasible to enumerate all negative solutions. For example, for the reflexive property (graphs in which all nodes have a self-loop) with a scope of 6, there are only around 1 Million graphs (1048576) that satisfy the property out of the 68.7 Billion (2^{36}) possible graphs. We use Alloy to enumerate all positive solutions and then create an equal number of negative solutions. To create the set of negative solutions, we first create a random solution and verify if it violates the property using the Alloy evaluator. If the solution is a graph satisfying the property then it is discarded otherwise it is added to the set of negative solutions. Each solution has a feature vector and a binary label (1 for the positive class and 0 for the negative class). The features are represented using an adjacency matrix. For example, if the scope of the reflexive property is 6, then a 36 bit vector is used. The smallest scope was choosen for each property such that there are $\geq$ 90,000 positive solutions.

Although, Alloy was used in the generation of datasets, the generated datasets are not dependent on Alloy. Please note that for positive solutions, all possible solutions were generated. The same set of solutions will be generated by any SAT solver (Representation format can be different). For negative solutions, we randomly created them and confirmed them using the Alloy evaluator which

[3] https://github.com/mcmlplus/mcmlplus

Table 1: Quantifying the learnability of Decision Trees on graph properties with *ProjMC*. Metrics from *QuantifyML*, *MCML* and Statistical calculation on test set(*Stat*). "-" indicates a time-out of 5000 seconds.

Property	*Accuracy*			*Precision*			*Recall*			*F1-Score*		
	Stat	*MCML*	*QuantifyML*	*Stat*	*MCML*	*QuantifyML*	*Stat*	*MCML*	*QuantifyML*	*Stat*	*MCML*	*QuantifyML*
Antisymmetric	0.9997	0.9996	0.9996	0.9996	0.9938	0.9938	0.9998	0.9998	0.9998	0.9997	0.9968	0.9968
Connex	0.9957	0.9934	0.9934	0.9933	0.2106	0.2106	0.9982	0.9984	0.9984	0.9957	0.3478	0.3478
Equivalence	0.9997	-	-	0.9994	-	-	1.0000	1.0000	1.0000	0.9997	-	-
Irreflexive	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
NonStrictOrder	0.9990	0.9984	-	0.9985	0.0012	-	0.9994	0.9995	0.9995	0.9990	0.0024	-
PartialOrder	0.9934	0.9877	0.9877	0.9879	0.2473	0.2473	0.9991	0.9992	0.9992	0.9935	0.3964	0.3964
PreOrder	0.9985	0.9973	-	0.9974	0.0011	-	0.9996	0.9996	0.9996	0.9985	0.0022	-
Reflexive	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
StrictOrder	0.9988	0.9978	-	0.9979	0.0009	-	0.9997	0.9998	0.9998	0.9988	0.0018	-
TotalOrder	0.9999	-	-	0.9997	-	-	1.0000	1.0000	-	0.9999	-	-
Transitive	0.9999	0.9764	0.9764	0.9997	0.1622	0.1622	1.0000	0.9913	0.9913	0.9999	0.2789	0.2789

Table 2: Quantifying the learnability of Decision Trees on graph properties with *ApproxMC*. *Diff* shows the difference between MCML and QuantifyML metrics.

Property	*Accuracy*			*Precision*			*Recall*			*F1-Score*		
	MCML	*QuantifyML*	*Diff*	*MCML*	*QuantifyML*	*Diff*	*MCML*	*QuantifyML*	*Diff*	*MCML*	*QuantifyML*	*Diff*
Antisymmetric	0.9996	0.9996	0.0000	0.9935	0.9936	-0.0001	0.9998	0.9998	0.0000	0.9967	0.9967	0.0000
Connex	0.9935	0.9936	-0.0001	0.2258	0.2292	-0.0032	0.9985	0.9985	0.0000	0.3683	0.3728	-0.0045
Equivalence	0.9995	0.9995	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000
Irreflexive	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
NonStrictOrder	0.9983	0.9985	-0.0002	0.0011	0.0012	-0.0001	0.9995	0.9995	0.0000	0.0022	0.0024	-0.0002
PartialOrder	0.9864	0.9880	-0.0016	0.2407	0.2535	-0.0128	0.9992	0.9992	0.0000	0.3879	0.4045	-0.0166
PreOrder	0.9972	0.9973	-0.0001	0.0012	0.0012	0.0000	0.9997	0.9997	0.0000	0.0024	0.0024	0.0000
Reflexive	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
StrictOrder	0.9979	0.9979	0.0000	0.0009	0.0010	-0.0001	0.9998	0.9998	0.0000	0.0019	0.0020	-0.0001
TotalOrder	0.9997	0.9997	0.0000	0.0000	0.0000	0.0000	1.0000	0.0017	0.9983	0.0000	0.0000	0.0000
Transitive	0.9760	0.9773	-0.0013	0.1588	0.1826	-0.0238	0.9902	0.9922	-0.0020	0.2737	0.3085	-0.0348

Table 3: Efficiency comparison QuantifyML vs. MCML. Number of primary variables (*Prim. Vars*) are same for MCML and QuantifyML, Total number of *Clauses* for true positive (TP), false negative (FN), false positive (FP), and true negative (TN) in the CNF formulae. Times taken by ApproxMC and ProjMC model counters. "-" indicates time-out of 5000 seconds

Property	*Prim*	*TP Clauses*		*FN Clauses*		*FP Clauses*		*TN Clauses*		*Time (s) - projMC*		*Time (s) - ApproxMC*	
	Vars	*MCML*	*QuantifyML*	*MCML*	*QuantifyML*	*MCML*	*QuantifyML*	*MCML*	*QuantifyML*	*MCML*	*QuantifyML*	*MCML*	*QuantifyML*
Antisymmetric	25	1015	86364	1031	86540	1185	86427	1201	86603	1.5	121.5	2.1	15.8
Connex	25	69	7643	74	7508	231	4301	236	4236	0.3	4.8	0.8	0.4
Equivalence	100	2591	44644	2582	44617	3349	40562	3340	40546	-	-	34.1	30.1
Irreflexive	25	10	172	6	208	147	142	143	162	0.0	0.0	0.5	0.2
NonStrictOrder	36	507	20336	489	20520	624	14303	606	14424	12.2	-	1.9	4.6
PartialOrder	25	514	23563	435	23674	576	22399	497	22499	0.8	1404.7	1.2	2.9
PreOrder	36	528	22850	516	22892	659	17158	647	17188	34.5	-	2.0	3.8
Reflexive	25	10	227	6	207	98	177	94	165	0.0	0.0	0.5	0.1
StrictOrder	36	480	16924	464	17212	611	12647	595	12782	14.1	-	1.9	2.8
TotalOrder	81	1834	39521	1819	39778	2331	42961	2316	43033	-	-	8.4	8.9
Transitive	25	774	43456	672	43534	862	43544	760	43622	14.9	2377.5	2.0	8.4

simply evaluates if the given solution is positive or negative. Therefore, there is no bias or dependence in using Alloy.

4.1.2 Decision Trees. Scikit-Learn Library was used to implement decision tree models. The quality of the split (criterion) was measured using *gini* and rest of the hyper-parameters were set to default settings. Experiments were also performed using hyper-parameter tuning but results were only slightly improved with a huge increase in time. QuantifyML uses a time-out of 5000 seconds which is common in field of model counting.

4.1.3 Results. Table 1 presents the *QuantifyML* and *MCML* metrics (accuracy, precision, recall and F1-score) for the decision-tree models for the respective properties. These were calculated using the ProjMC model counter (Section 2) which is an exact model counter. Please note that we applied both techniques to the same model for each property. As can be seen , the values of the metrics obtained by *QuantifyML* and *MCML* match exactly for all the properties. This is as expected since both the tools seek to obtain precise quantification over the input space. *QuantifyML* is more general and uses a C program representation of any machine learning model and the CBMC tool to perform the translation to CNF form, whereas *MCML* has an implementation that is dedicated to decision-trees. These results serve as a check for the correctness of *QuantifyML* and show that the tool produces exactly the same results as the technique that is dedicated to decision-trees.

However, *QuantifyML* is less efficient than *MCML*. Table 3 shows the times taken by *QuantifyML* and *MCML* using ProjMC (*Time(s)*

projMC column). It can be observed that both *MCML* and *QuantifyML* time out (after 5000 seconds) for TotalOrder and Equivalence properties. *QuantifyML* times out for the following properties as well; NonStrictOrder, StrictOrder and PreOrder. This is because the CNF formulas generated by the CBMC tool after the analysis of the C program representation of the machine learning model is larger than that produced by *MCML*, which has a custom implementation for decision-trees. Table 3 shows the size of the CNF formulas in terms of primary variables and clauses. ProjMC is an exact model counter and takes a long time to process formulas with large number of clauses.

To alleviate this scalability problem, we experimented with another model counter, namely ApproxMC (Section 2), which is faster but produces approximate counts. The column *Time(s) ApproxMC* in Table 3 shows the times and Table 2 presents the respective metrics for both *QuantifyML* and *MCML* using ApproxMC. As it can be observed, the analysis times for *QuantifyML* are greatly reduced and we are able to obtain results for all the properties. Due to the approximate nature of the model counter, there are minor differences in the metrics computed with *QuantifyML* vs. *MCML*. However, these differences are minor (as can be seen in the *Diff* columns of the table). In some cases, *QuantifyML* is closer to the true values (presented in table 1), while in others *MCML* is closer, however, the approximate results are within +/- 15% of the true results. [33] empirically determined ApproxMC model counts to be +/- 16% of the true counts.

In order to understand the benefits of *QuantifyML* in assessing the learnability of models, we considered the metrics calculated statistically on a test set. We present the statistical metrics (accuracy, precision, recall and F1-scores) for the decision-tree models in Table 1. It can be seen that for all the properties, the statistical accuracy values are higher than the true values. This difference is more pronounced for the precision metric. For instance, for the Connex, PartialOrder and Transitive properties, while the statistical precision values are greater than 98%, the actual precision of these models are less than 25%. This highlights that statistical metrics could be mis-leading and give a false impression of good performance. The F1-score indicates the generalizability of the models. It can be seen that specifically for the above mentioned three properties, the true F1-scores are quite poor indicating that the models are possibly over-fitted to the respective train sets. However, this fact would be hidden if we rely only on the statistical metrics. We would like to highlight that even with approximate results obtained using ApproxMC, *QuantifyML* metrics still present a more accurate picture of the accuracy and generalizability of the models in comparison with the statistical metrics. Specifically, for the Equivalence and TotalOrder properties (Table 2), we can see that the number of false positives is very high leading to very low values for the true precision. The true F1-scores for these models are close to 0, however, the respective statistical F1-scores are close to 0.99, giving a false impression of good performance.

Tables 4 and 5 present *QuantifyML* and statistically calculated metrics for decision-tree and neural network models respectively for relational properties of small (4-node) graphs. We describe these models in more detail in the next section. We can observe a similar trend here as well that the statistical metrics indicate that the models have high accuracy and F1-scores for all properties. The decision-tree models (Table 4) for the Antisymmetric, Irreflexive, Reflexive, NonStrictOrder, PartialOrder and PreOrder properties have statistical accuracy and the F1-scores of 100%. However, the precise quantification by *QuantifyML* highlights that for the NonStrictOrder and PreOrder properties, the models in fact have less than 100% accuracy and more importantly have poor precision indicating large number of false positives. The decision-trees for StrictOrder property seems to have the lowest accuracy and F1-score when calculated statistically. However, the *QuantifyML* scores highlight that this is mis-leading and in truth the decision-tree for the Connex Property has the lowest accuracy and F1-score. For the neural network models (Table 5), in all cases except Irreflexive and Reflexive properties, the accuracy values calculated using *QuantifyML* highlight that the true performance is mostly worse and in some cases better (PartialOrder, Transitive) than the respective statistical accuracy metric values. The precision and recall metrics seem to suffer greatly when calculated purely statistically (as can be seen from the *Diff* columns in the table 5). This indicates that they give a false impression of good generalizability of the respective models, while in truth the F1-scores are less than 50% for most of the properties (refer *F1-Score* column in table 5).

4.2 RQ 2: Comparing the performance of different models.

For this set of experiments, we seek to evaluate *QuantifyML* on different models that are trained on the same problem.

Consider the problem of learning relational properties of graphs as explained earlier in *RQ 1*. Although this seems a fairly simple problem with binary decisions and binary input features, it is not immediately apparent which learning algorithm would work best to learn a suitable classifier. The properties range from simple ones such as Reflexive to more complex ones such as StrictOrder and Connex (ref supplementary material - section 1). Each property translates to a different set of features. While a decision-tree classifier may be faster to learn and easier to interpret, it may be overfitted. On the other hand, employing a deep neural network algorithm which relies on feature extraction may in fact not yield good results for this problem considering the inputs are purely binary.

We applied the two different learning algorithms, decision-trees and neural networks, to learn classification models for the same set of properties using the same dataset for training. We were unable to apply QuantifyML to analyze neural network models with greater than 16 features due to the limitation in the scalability of the model counters (both ApproxMc ProjMC were considered). Therefore we restricted the size of the graphs to have 4 nodes. The model counter times out after 10 hours when considering graphs with more than 4 nodes for all properties. However, with the reduction in the number of nodes in the graph, the number of *positive* graphs become very low for the Equivalence and TotalOrder properties. This hinders the learning of suitable classifiers. Therefore we consider only the remaining 9 properties for this study.

4.2.1 Architecture. The neural networks had two dense layers. The first layer had 3 nodes with a *linear* activation function and the second layer had 2 nodes with a *softmax* activation function. The neural networks were trained upto 50 epochs with a batch size of 10. Adam optimizer was used. For the decision trees, the quality

Table 4: Quantifying the learnability of Decision Trees on graph (4-node) properties with *ProjMC*. *Diff* shows the difference between Statistical (*Stat*) and *QuantifyML* metrics.

Property	*Accuracy*			*Precision*			*Recall*			*F1-Score*		
	Stat	*QuantifyML*	*Diff*	*Stat*	*QuantifyML*	*Diff*	*Stat*	*QuantifyML*	*Diff*	*Stat*	*QuantifyML*	*Diff*
Antisymmetric	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
Connex	0.9932	0.8179	-0.1752	0.9865	0.4219	-0.5646	1.0000	0.0625	-0.9375	0.9932	0.1089	-0.8843
Irreflexive	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
NonStrictOrder	1.0000	0.9721	-0.0279	1.0000	0.1069	-0.8931	1.0000	1.0000	0.0000	1.0000	0.1932	-0.8068
PartialOrder	0.9957	0.9919	-0.0038	0.9916	0.8690	-0.1226	1.0000	1.0000	0.0000	0.9958	0.9299	-0.0659
PreOrder	1.0000	0.9693	-0.0307	1.0000	0.1499	-0.8501	1.0000	1.0000	0.0000	1.0000	0.2607	-0.7393
Reflexive	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
StrictOrder	0.9545	0.9721	0.0175	0.9200	0.1069	-0.8131	1.0000	1.0000	0.0000	0.9583	0.1932	-0.7651
Transitive	0.9850	0.9799	-0.0051	0.9810	0.7524	-0.2285	0.9904	0.9990	0.0086	0.9856	0.8583	-0.1273

Table 5: Quantifying the learnability of Neural Network on graph (4-node) properties with *ProjMC*. *Diff* shows the difference between Statistical (*Stat*) and *QuantifyML* metrics.

Property	*Accuracy*			*Precision*			*Recall*			*F1-Score*		
	Stat	*QuantifyML*	*Diff*	*Stat*	*QuantifyML*	*Diff*	*Stat*	*QuantifyML*	*Diff*	*Stat*	*QuantifyML*	*Diff*
Antisymmetric	0.8058	0.7614	-0.0445	0.7520	0.4211	-0.3309	0.9095	0.9093	-0.0002	0.8233	0.5756	-0.2476
Connex	0.9658	0.7866	-0.1791	0.9359	0.2326	-0.7033	1.0000	0.0865	-0.9135	0.9669	0.1261	-0.8408
Irreflexive	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
NonStrictOrder	0.9773	0.9054	-0.0719	0.9583	0.0338	-0.9245	1.0000	0.9909	-0.0091	0.9787	0.0654	-0.9133
PartialOrder	0.7803	0.8303	0.0500	0.8367	0.2002	-0.6364	0.7051	0.7260	0.0210	0.7652	0.3139	-0.4514
PreOrder	0.9577	0.8825	-0.0753	0.9302	0.0433	-0.8870	1.0000	0.9803	-0.0197	0.9639	0.0829	-0.8810
Reflexive	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000	1.0000	1.0000	0.0000
StrictOrder	0.9545	0.9409	-0.0136	0.9200	0.0535	-0.8665	1.0000	1.0000	0.0000	0.9583	0.1016	-0.8567
Transitive	0.7722	0.7903	0.0181	0.8063	0.1864	-0.6198	0.7404	0.7258	-0.0145	0.7719	0.2967	-0.4753

of the split (criterion) was measured using *gini* and rest of the hyper-parameters were set to default settings.

4.2.2 Results. We applied *QuantifyML* to obtain the accuracy, precision, recall and F1-score metrics for the models learnt using both the techniques. These metrics were also calculated statistically on a test set (same for both the learning algorithms). Tables 4 and 5 present the results.

The models for the Irreflexive and Reflexive properties have 100% accuracy and F1-score, both when learnt using the decision-tree learning algorithm or as a neural network. These two are indeed very simple graph properties. However, decision-tree models have the ability to learn more complex properties such as Antisymmetric and StrictOrder as well. Overall the decision-tree models seem to have better accuracy and generalizability than the respective neural network models. Note, that while such a comparison can be done using the statistical metrics, their lack of precision may lead to wrong interpretations. For instance, for the StrictOrder property, the statistical accuracy, precision, recall and F1-scores are exactly the same for the neural network and decision-tree models, however, the corresponding *QuantifyML* metrics highlight that for this problem, the decision-tree models are in fact better than neural networks in terms of the true performance.

Albeit using a simple example, this study highlights the benefit of *QuantifyML* in enabling one to compare the true performance of different models, and different learning algorithms, for a given problem. Note that as long as the models being compared are for the same application, they need not have been trained using the same set of data points.

4.3 RQ 3: Quantifying the robustness of image classifiers.

We use the MNIST data set (Modified National Institute of Standards and Technology data set), which is a large collection of handwritten digits that is commonly used for training various image processing systems [34]. MNIST has 60,000 training input images, each comprised of 784 pixels and belonging to one of 10 labels (digits 0 through 9). We trained a decision-tree model on this dataset. For the decision trees, the quality of the split (criterion) was measured using *entropy* and rest of the hyper-parameters were set to default settings. The overall accuracy of this model on the test set was 83.64%. The size of the CNF files for this problem is large (refer supplementary material - Table 5). This can be attributed to the 784 integer input variables and the logic for image classification which results in bigger trees. There are 25088 primary variables and around 31 million clauses in the CNF formulas for each label. It was not feasible to use the exact model counter ProjMC , therefore, we used the ApproxMC model counter for this study. Typically neural networks are used for the problem of image classification. However, both ApproxMC and ProjMC were not scalable enough to handle the CNF files generated from the neural networks that we trained for this study.

4.3.1 Results. We selected (randomly) an image for each of the 10 labels to evaluate the robustness of the model in the neighborhood of each of these inputs. We considered regions around these inputs for $\epsilon = 1$; these represent all the inputs that can be generated by altering each pixel of the given image by +/- 1. Column *Total count*$_\epsilon$ in Table 6 shows the total number of images in the $\epsilon = 1$ neighborhood of each input. We then employ *QuantifyML* to precisely quantify the number of inputs within the $\epsilon = 1$ neighborhood that are given the correct label. In order to obtain this count, for every

Table 6: Quantifying robustness for the MNIST model.

Input	*Actual Label*	*Total count*$_\epsilon$	*Correctly classified count*$_\epsilon$	*Robustness*$_\epsilon$ *(%)*	*Accuracy*$_\epsilon$*(100)*	*Accuracy*$_\epsilon$*(1000)*	*Accuracy*$_\epsilon$*(10000)*	*Accuracy (Test Set)*
1	0	3.32E+270	2.21E+270	66.67	65.00	67.90	67.05	92.65
2	1	9.59E+247	3.15E+247	32.81	34.00	32.10	32.66	96.12
3	2	3.53E+258	8.81E+257	25.00	32.00	25.90	25.30	83.91
4	3	1.92E+272	1.92E+272	99.99	93.00	93.70	93.62	77.52
5	4	3.42E+264	3.42E+264	99.99	52.00	61.50	63.58	82.08
6	5	4.02E+259	4.02E+259	99.99	100.00	100.00	100.00	76.23
7	6	1.04E+258	5.22E+257	50.00	47.40	47.40	50.21	85.39
8	7	1.17E+262	1.17E+262	99.99	100.00	100.00	100.00	85.60
9	8	9.99E+266	9.99E+266	99.99	100.00	100.00	100.00	73.72
10	9	1.84E+253	6.89E+252	37.50	38.20	38.20	38.12	80.67

input, we apply *QuantifyML* to quantify the portion of the input region that leads to the same label as the original input. Column *Correctly classified count*$_\epsilon$ in Table 6 shows this count for each input. The corresponding *Robustness*$_\epsilon$ value shows the accuracy with which the model classifies the inputs in the region to the same label.

The results indicate that the robustness of the model is poor or the model is more vulnerable to attacks around the inputs corresponding to labels 1, 2 and 9 respectively. Note also that in principle, *QuantifyML* could be used to provide proofs of robustness (if *Robustness*$_\epsilon$*=100%)*; however we could not obtain such proofs here.

We also computed an accuracy metric statistically by perturbing each image within ϵ to randomly generate sample sets of size 100, 1000 and 10000 images respectively. We then executed the model on each set to determine the corresponding labels and computed the respective accuracies as shown in columns *Accuracy*$_\epsilon$*(100)*, *Accuracy*$_\epsilon$*(1000)*, and *Accuracy*$_\epsilon$*(10000)*. The statistically computed accuracies are close to the *Robustness*$_\epsilon$ values for most of the labels. We can see that for labels 5,7 and 8, the statistical accuracies with all sample sets are 100% respectively. This gives a false impression of adversarial robustness around these inputs. The corresponding *Robustness*$_\epsilon$ of 99.99% indicates that there are subtle adversarial inputs which get missed when the robustness is determined statistically. We can also observe an interesting case, for the input corresponding to label 4, where the sample sets considered for statistical accuracy seem to include a number of inputs that are incorrectly classified. *Robustness*$_\epsilon$ of 99.99% indicates that only 0.01% of the inputs are incorrectly classified. However, these inputs comprise of almost 36% of the samples considered for statistical accuracy. This seems to indicate that this region is more vulnerable to attacks than the inputs corresponding to labels 5 and 8, while in truth, these regions are very similar in terms of robustness. It may be the case that this result for *Robustness*$_\epsilon$ is incorrect, which may be due to the approximation in the model counter. We plan to investigate this issue further.

The last column, *Accuracy (Test Set)* in Table 6, shows the accuracy of the model per label when evaluated statistically on the whole MNIST test set. Note that although the model may have high statistical accuracy, it can have low adversarial robustness. Consider the input for label 1, although the statistical targeted accuracy is 96%, the *Robustness*$_\epsilon$ around the input is only around 32% indicating poor adversarial robustness.

We would like to highlight that although the above discussion is based on the results produced by ApproxMC which produces approximate counts, it has been shown in an earlier study [33] that the outputs are typically within 16% of the outputs from ProjMC (exact model counter). The results obtained using statistical measures typically have a much higher difference from the exact counts (as was highlighted in RQ1).

5 RELATED WORK

5.1 Quantitative Analysis of ML models

Quantitative analysis of neural networks has typically been performed using statistical methods to obtain probabilistic guarantees. For instance, VeriFair is an approach which performs probabilistic verification of fairness properties by leveraging sampling and concentration inequalities [35]. PROVEN [36] is another framework for probabilistic analysis that employs existing robustness verification tools to derive probabilistic certificates for robustness of neural networks. PROVERO [37] performs quantitative verification to provide certificates for adversarial robustness by performing sampling of the inputs space and treating the model as a black-box. While statistical and sampling based approaches enable scalability, they do not provide precise quantification which is required to accurately assess the learnability and robustness of the models.

The technique in [38] uses symbolic analysis and constraint solution space quantification to precisely quantify probabilistic properties of neural networks, thereby enabling analysis of fairness and robustness of the models. FairSquare [39] is another probabilistic framework that implements a custom symbolic-volume-computation algorithm for fairness and bias analysis in neural networks. In contrast, our approach is not specialized only for neural networks and can be applied to assess the learnability and robustness of any machine learning model.

5.2 Model Counting Applications in ML

Model counting has been typically applied in machine learning for probabilistic reasoning [40–43]. Existing work that uses model counting technology for quantifying performance have been restricted for a sub-class of neural networks called Binarized Neural Networks (BNNs) [44]. The technique in [45] performs a translation of a BNN to SAT/CNF([46]) and use model-counters to provide quantitative estimates for the satisfaction of logical properties, which enables applications such as providing adversarial robustness guarantees and evaluating the efficacy of trojan attacks. In contrast, *QuantifyML* facilitates applications beyond binarized neural networks, albeit small. Expressing the model as a C program enables utilizing standard bounded model checking technology for C programs, and the advancements in this field [47, 48] to perform the analysis.

As already mentioned, the work in [49] (MCML) presents an approach that applies model counting technology to perform quantitative assessment of the performance of decision-tree classifier models. The *MCML* tool is limited to binary decision trees and has been used on decision-tree models when used to learn relational properties of graphs. *QuantifyML* goes beyond this approach and enables quantification of the performance of any type of machine learning model, with non-binary inputs and multi-class outputs. Our evaluation presents applications such as robustness analysis of decision-tree models on an image-classification problem (MNIST), comparison of neural network and decision-tree models for learning relational properties of graphs, which cannot be achieved by using the *MCML* tool.

5.3 Analyzing Learnability

Many frameworks based on statistical learning theory [50, 51] have analyzed the learnability of machine learning models. Popular ones such as the Probably Approximately Correct (PAC) learning framework [52] enable formal analytical characterization of the number of examples needed to train models for binary classification tasks. The work in [53] provides a detailed set of conditions for learnability in terms of the VC (Vapnik- Chervonenkis (VC) theory [54]) dimension. QuantifyML complements theoretical frameworks by providing empirical evidence to precisely quantify generalizability both in terms of the accuracy as well as precision, recall and F1-Score with respect to a given ground truth.

6 LIMITATIONS, FUTURE WORK AND CONCLUSION

6.1 Limitations and Future Work

The studies presented in the section 4 highlight the benefits of using *QuantifyML* to quantify the learnability and robustness of machine learning models. However, one of the main limitations of *QuantifyML* is scalability, which is mainly inhibited by the scalability of the model counting technology. While approximate model counting is more efficient, it is less precise and still could not scale to large models. We present below a discussion analyzing the reasons for the cases that *QuantifyML* was unable to handle and possible remedies.

We found that the analysis of neural network models was specifically challenging. The model counters (both exact and approximate) timed out while analyzing the neural network models for learning graph properties when the size was greater than 4 nodes. For the image classification study, *QuantifyML* could not handle neural network models, despite the following attempts to reduce the state space. We represented the weights and biases as *longs* instead of *floats*, and considered perturbing only a subset (10 out of 784) pixels. However, although this did reduce the number of clauses and variables in the CNF files, the model counters still timed out. One of the reasons for this poor performance is the difference in the architecture of neural network models vs. decision-trees. Decision-tree learning produces a compact classifier which uses only a subset of the input features. For example, for the reflexive property, the decision tree learned that all diagonal inputs should be 1. Therefore, the decision tree only checks 5 diagonal input variables instead of 25 variables. On the other hand, in a neural network, all the input variables are considered and there are more complex operations performed on them. This results in a larger CNF with more number of variables and clauses for the neural network models. More details on the size of the CNF files for neural network models and for the MNIST robustness study can be found in the supplementary material.

A possible solution to alleviate the scalability problem is to use methods such as slicing to enable parallel, and/or compositional analysis of the C program representation of the models. Identifying important input features (using attribution techniques or decision-tree learning) and subsequently applying neural network learning on the reduced input feature set, could potentially yield smaller networks and enable their analysis using *QuantifyML*. We plan to investigate these ideas as part of future work. We also plan to add support for other decision trees and neural network learning libraries and for other machine learning algorithms such as Adaboost Trees [55], Gradient Boosting Trees [56], Random Forest Trees [57] and SVM [58].

6.2 Conclusion

This paper presented an approach, *QuantifyML*, which employs model counting to assess the *learnability* and *robustness* of machine learning models. Efficacy of machine learning models is usually evaluated on a test set but this could often be misleading. *QuantifyML* uses model counting technology to quantify the accuracy and generalizability of any machine learning model. We show that *QuantifyML* can evaluate the learnability of models by comparing the counts for the outputs to ground truth values. *QuantifyML* can also compare the performance of different models that may be built with different machine learning algorithms (e.g., decision-trees vs. neural networks). *QuantifyML* can also quantify the robustness of trained models around given inputs.

REFERENCES

[1] W.-J. Kuo, R.-F. Chang, D.-R. Chen, and C. C. Lee, "Data mining with decision trees for diagnosis of breast tumor in medical ultrasonic images," *Breast cancer research and treatment*, vol. 66, no. 1, pp. 51–57, 2001.

[2] J. Bastos, "Credit scoring with boosted decision trees," 2007.

[3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov 2012.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012, pp. 1097–1105.

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013, technical Report. http://arxiv.org/abs/1312.6199.

[6] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *EuroS&P*, 2016.

[7] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100270, 2020.

[8] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[9] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting," 2008.

[10] B. Bonakdarpour and S. S. Kulkarni, "Exploiting symbolic techniques in automated synthesis of distributed programs with large state space," in *ICDCS*, June 2007.

[11] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 266–280.

[12] M. Usman, W. Wang, M. Vasic, K. Wang, H. Vikalo, and S. Khurshid, "A study of the learnability of relational properties: Model counting meets machine learning (mcml)," ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1098–1111. [Online]. Available: https://doi.org/10.1145/3385412.3386015

[13] B. Demsky and M. C. Rinard, "Automatic detection and repair of errors in data structures," in *OOPSLA*, 2003.

[14] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (T)," in *ASE*, 2015.

[15] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "DeepGini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT ISSTA*, 2020, pp. 177–188.

[16] M. Sotoudeh and A. V. Thakur, "Correcting deep neural networks with small, generalizing patches," in *Workshop on Safety and Robustness in Decision Making*, 2019.

[17] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[18] C. Jin, L. De-Lin, and M. Fen-Xiang, "An improved id3 decision tree algorithm," in *2009 4th International Conference on Computer Science & Education*. IEEE, 2009, pp. 127–130.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[20] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 117–148, 2003.

[21] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, 2004, pp. 168–176.

[22] J.-M. Lagniez and P. Marquis, "A recursive algorithm for projected model counting," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1536–1543.

[23] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.

[24] M. Soos, "Cryptominisat v4," *SAT Competition*, vol. 23, 2014.

[25] R. A. Aziz, G. Chu, C. Muise, and P. J. Stuckey, "sat: Projected model counting," in *SAT*, 2015.

[26] M. Usman, W. Wang, and S. Khurshid, "Testmc: Testing model counters using differential and metamorphic testing," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 709–721.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[28] N. Ketkar, "Introduction to keras," in *Deep learning with Python*. Springer, 2017, pp. 97–111.

[29] D. Gopinath, G. Katz, C. S. Pasareanu, and C. Barrett, "DeepSafe: A data-driven approach for checking adversarial robustness in neural networks," *arXiv preprint arXiv:1710.00486*, 2017.

[30] M. Abbasi, A. Rajabi, C. Gagné, and R. B. Bobba, "Toward adversarial robustness by diversity in an ensemble of specialized deep neural networks," in *Advances in Artificial Intelligence*, C. Goutte and X. Zhu, Eds. Cham: Springer International Publishing, 2020, pp. 1–14.

[31] J. M. Cohen, E. Rosenfeld, and J. Z. Kolter, "Certified adversarial robustness via randomized smoothing," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019, pp. 1310–1320. [Online]. Available: http://proceedings.mlr.press/v97/cohen19c.html

[32] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, Apr. 2002.

[33] W. Wang, M. Usman, A. Almaawi, K. Wang, K. S. Meel, and S. Khurshid, "A study of symmetry breaking predicates and model counting," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 115–134.

[34] "The MNIST database of handwritten digits Home Page," http://yann.lecun.com/exdb/mnist/.

[35] O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness properties via concentration," *PACMPL*, vol. 3, no. OOPSLA.

[36] L. Weng, P. Chen, L. M. Nguyen, M. S. Squillante, A. Boopathy, I. V. Oseledets, and L. Daniel, "PROVEN: verifying robustness of neural networks with a probabilistic approach," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019, pp. 6727–6736. [Online]. Available: http://proceedings.mlr.press/v97/weng19a.html

[37] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena, "Scalable quantitative verification for deep neural networks," *CoRR*, 2020.

[38] H. Converse, A. Filieri, D. Gopinath, and C. S. Păsăreanu, "Probabilistic symbolic analysis of neural networks," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020.

[39] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "Fairsquare: probabilistic verification of program fairness," *PACMPL*, vol. 1, no. OOPSLA, pp. 80:1–80:30, 2017.

[40] M. Chavira and A. Darwiche, "On probabilistic inference by weighted model counting," *JAI*, vol. 172, no. 6-7, 2008.

[41] D. Fierens, G. V. den Broeck, I. Thon, B. Gutmann, and L. D. Raedt, "Inference in probabilistic logic programs using weighted cnf's," *CoRR*, vol. abs/1202.3719, 2012.

[42] R. Gens and P. Domingos, "Learning the structure of sum-product networks," in *ICML*, 2013.

[43] Y. Liang, J. Bekker, and G. V. den Broeck, "Learning the structure of probabilistic sentential decision diagrams," in *UAI*, 2017.

[44] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *NIPS*, 2016.

[45] T. Baluta, S. Shen, S. Shinde, K. S. Meel, and P. Saxena, "Quantitative verification of neural networks and its security applications," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, 2019, pp. 1249–1264.

[46] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," in *AAAI*, 2018.

[47] S. Falke, F. Merz, and C. Sinz, "The bounded model checker llbmc," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[48] A. Jana, U. P. Khedker, A. Datar, R. Venkatesh, and C. Niyas, "Scaling bounded model checking by transforming programs with arrays," 2016.

[49] M. Usman, W. Wang, M. Vasic, K. Wang, H. Vikalo, and S. Khurshid, "A study of the learnability of relational properties: model counting meets machine learning (mcml)," *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[50] V. N. Vapnik, *The Nature of Statistical Learning Theory*, 1995.

[51] S. Shalev-Shwartz, O. Shamir, N. Srebro, and K. Sridharan, "Learnability, stability and uniform convergence," *JMLR*, vol. 11, Dec. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1953019

[52] L. G. Valiant, "A theory of the learnable," *CACM*, vol. 27, no. 11, Nov. 1984. [Online]. Available: http://doi.acm.org/10.1145/1968.1972

[53] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Learnability and the Vapnik-Chervonenkis dimension," *JACM*, vol. 36, no. 4, 1989.

[54] V. N. Vapnik and A. Ya. Chervonenkis, "On the uniform convergence of relative frequencies of events to their probabilities," *Theory of Probability and its Applications*, 1971.

[55] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119 – 139, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002200009791504X

[56] J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, 10 2001. [Online]. Available: https://doi.org/10.1214/aos/1013203451

[57] T. K. Ho, "Random decision forests," in *Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, 1995.

[58] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995. [Online]. Available: https://doi.org/10.1007/BF00994018