International Journal on Software Tools for Technology Transfer (2020) 22:601–615
https://doi.org/10.1007/s10009-020-00577-w

STTT

Special Issue: SPIN 2019

A study of learning likely data structure properties using machine learning models

Muhammad Usman[1] · Wenxi Wang[1] · Kaiyuan Wang[1] · Cagdas Yelen[1] · Nima Dini[1] · Sarfraz Khurshid[1]

Published online: 7 June 2020
© Springer-Verlag GmbH Germany, part of Springer Nature 2020

Abstract
Data structure properties are important for many testing and analysis tasks. For example, model checkers use these properties to find program faults. These properties are often written manually which can be error prone and lead to false alarms. This paper presents the results of controlled experiments performed using existing machine learning (ML) models on various data structures. These data structures are dynamic and reside on the program heap. We use ten data structure subjects and ten ML models to evaluate the learnability of data structure properties. The study reveals five key findings. One, most of the ML models perform well in learning data structure properties, but some of the ML models such as *quadratic discriminant analysis* and *Gaussian naive Bayes* are not suitable for learning data structure properties. Two, most of the ML models have high performance even when trained on just 1% of data samples. Three, certain data structure properties such as *binary heap* and *red black tree* are more learnable than others. Four, there are no significant differences between the learnability of varied-size (i.e., up to a certain size) and fixed-size data structures. Five, there can be significant differences in performance based on the encoding used. These findings show that using machine learning models to learn data structure properties is very promising. We believe that these properties, once learned, can be used to provide a run-time check to see whether a program state at a particular point satisfies the learned property. Learned properties can also be employed in the future to automate static and dynamic analysis, which would enhance software testing and verification techniques.

Keywords Data structure invariants · Machine learning · Korat · Learnability

1 Introduction

Data structure invariants play an extensive role in the verification and validation of software systems. These are properties that data structures should satisfy, and are also termed as class invariants in object-oriented programming [39,45]. For example, a binary tree should satisfy properties such that there is no cycle in the tree and that all nodes are reachable from the root node.

We believe that data structure invariants play a vital role in the field of software testing. Model checkers usually use these invariants as assertions and try to violate these invariant properties in order to find bugs in programs. If an invariant is violated, it exposes a bug in the software system [25,44,67]. Data structure repair techniques use these invariants for error recovery [13,18]. Many automated test generation tools [4, 38] employ these invariants to serve as test assertions.

Data structure invariants are often manually written which is why they are error prone. Hence, it is crucial to have a testing mechanism to provide a high confidence about the correctness of data structure invariants. Prior techniques used deep semantic analysis [9,32,47,52,55,69] and static and dynamic analysis [11,14,19,34,40,43,47,52,55,56,69] to solve the problem of testing data structure invariants.

✉ Muhammad Usman
muhammadusman@utexas.edu

Wenxi Wang
wenxiw@utexas.edu

Kaiyuan Wang
kaiyuanw@utexas.edu

Cagdas Yelen
cagdas@utexas.edu

Nima Dini
nima.dini@utexas.edu

Sarfraz Khurshid
khurshid@utexas.edu

[1] University of Texas at Austin, Austin, TX 78712, USA

While prior approaches showed promising results, we believe machine learning (ML) is another direction to solve this problem. To the best of our knowledge, few people have worked on the learnability of data structure properties [21,41]. We hypothesize that ML models can learn these properties and be used to test a data structure invariant, or find bugs in its implementation.

This paper presents a controlled empirical study of applying ten existing ML models to learn ten widely used data structure properties. These ML models include decision tree [51], ensemble tree classifiers, i.e., random forest tree [30], gradient boosting tree [23] and Adaboost tree [22], support vector machine [10], multi-layer perceptron [48], k-nearest neighbor [1], Gaussian naive Bayes [53], logistic regression classifier [2] and quadratic discriminant analysis [68]. Thus, this study uses a range of ML models. A variety of data structure subjects are also used so that we can see the generalization of ML models in learning data structure properties. These models have also been used in some recent studies [4,18,21]. Data structure subjects used in this study include binary heap, binary search tree, binary tree, directed acyclic graph, disjoint set, Fibonacci heap, heap array, red black tree, sorted list and singly linked list.

A state-of-the-art test input generation tool named Korat [4,37] was used to create datasets for training and testing. Korat is written in Java, and each of the data structure invariants has a *repOK* predicate which is a check for properties of each structure. For example, for a binary search tree, *repOK* checks whether there are no cycles in the tree and all the elements are in the correct order. Korat takes as input a size parameter ϕ which sets the maximum size of the generated data structure. For example, if ϕ is `10`, it generates all possible binary search trees with up to `10` nodes.

Our study methodology is as follows. For each data structure invariant, we first use Korat to generate four datasets, namely α, β, γ and δ. For α, we only generate data structures with exact size of ϕ. For β, we generate data structures with sizes ranging from `1` to ϕ. Datasets γ and δ are like α and β, respectively, except that we convert them to one-hot encoding format (*OHE*) [28]. We provide detailed explanation of selecting the size ϕ in our evaluation (Sect. 3.1). We use Korat to generate all non-isomorphic valid and invalid data structures. The number of invalid structures is significantly larger than the number of valid structures because there are only a limited number of valid program states whereas there can be many invalid program states. For example, given a red black tree with `10` nodes, Korat generated 32,074,894 invalid red black trees and only 17,160 valid red black trees. This data imbalance is problematic because a naive ML model will learn to classify all instances as negative and still yield high accuracy. Therefore, we use the undersampling technique [29] to balance the datasets.

The size of the train and test data also affects the accuracy of ML models. Note that it might be infeasible to generate all positive and negative samples for a data structure invariant with a large size. For example, to generate all valid and invalid binary trees with `20` nodes, Korat has to explore a state space of 2^{180} candidates. To make our experiments feasible, we train and evaluate the performance of ML models using datasets with a manageable size (between `5` and `10` nodes). We performed all experiments with five different train–test ratios, including 75:25, 50:50, 25:75, 10:90 and 1:99. It was interesting to see how the ML models perform when trained on just 1% of dataset and tested on 99% of the dataset. To the best of our knowledge, this is the first work to study the performance of ML models using a train–test ratio of 1:99. We performed experiments to compare the learnability of varied-size data structures with the learnability of fixed-size data structures. In addition, we also studied if converting Korat's default representation to a one-hot encoding format affects the learnability of data structure properties.

In summary, this paper extends the previous work [64] in four directions:

- This study includes experiments with four additional ML models including quadratic discriminant analysis, Gaussian naive Bayes, k-nearest neighbor and logistic regression classifier.
- This study extends previous experiments with an extreme train–test ratio of 1:99. The results show that it is possible to train ML models on a small subset of a dataset and still achieve high performance.
- This work compares the learnability of varied-size data structures with the learnability of fixed-size data structures.
- This work studies the impact on the performance of using one-hot encoding representation, compared with Korat's default encoding.

Our study revealed five key findings:

- While most ML models are highly accurate in learning data structure properties, some ML models are not suitable for learning data structure properties, i.e., *Gaussian naive Bayes (GNB)* and *quadratic discriminant analysis (QDA)*.
- Decreasing the train–test ratio from 75:25 to 1:99 reduces accuracy by at most 6%. This result opens the possibility of learning data structure properties of larger sizes.
- Certain data structure properties are more learnable than others, e.g., *binary heap* and *red black tree*.
- There is no significant difference between the learnability of varied- and fixed-size data structures.

- One-hot encoding representation hinders the performance of varied-size data structures but improves the performance of fixed-size data structures.

The results of our extensive study show the feasibility of using ML models to learn data structure properties. Test generation tools can generate data structure instances and simply use trained ML models to verify if these instances are valid. We believe that verification using trained ML models will be more efficient, compared to executing numerous checks on each data structure instance. Thus, the use of ML models in software analysis holds a promising future, and we hope that machine learning could be employed for enhancing software reliability.

2 Background

This section introduces Korat and the ten ML models used in our study. It also explains data structure encoding format.

2.1 Korat

Korat is an automated test generation tool which generates all possible inputs given a *repOK* method, finitization on input domain and a Java predicate. Korat executes *repOK* for each possible data structure and filters them into valid (for which *repOK* returns `true`) and invalid structures (for which *repOK* returns `false`). Korat is efficient in that it prunes the search space by looking at the field accesses while still keeping the search complete and correct. Korat also generates only non-isomorphic structures, thus reducing the number of generated data structures.

To illustrate, Fig. 1 shows the `SinglyLinkedList` (SLL) class, including the `repOk` predicate and finitization `finSinglyLinkedList`. The SLL has a header field of type `Node` and a `size` field of type integer. The `Node` class declares an `element` field and a `next` field, representing the value of node and node next to it. The method `repOk` returns `true` if the list does not have any cycle and has the correct value for size, `false` otherwise. The finitization method `finSinglyLinkedList` specifies a bound on the total number of nodes, and the minimum and maximum values for size.

Korat internally represents each candidate structure using a candidate vector of integer indices. The vector length represents the number of object fields to search, and it depends on the finitization. To illustrate, for a finitization of up to `3` nodes (Node `1`, Node `2` and Node `3`) and size equal to `3`, Korat creates a candidate vector of length `8`: Index `0` represents the value of the header field; index `1` represents the size (and its value is fixed as `0` since the size is restricted to be `3`); and indexes `2` and `3` represent the integer value and the next node of Node `1`, respectively; likewise, indexes `4`, `5` and `6`, `7` represent the value and next node of Node `2` and Node `3`, respectively. The value of each index represents a node range from `0` to `3`, representing four possibilities: [`null`, Node `1`, Node `2` and Node `3`]. The list header has four possible values, and the list size is fixed to 1. The value and next node fields of each of the three nodes have four possible values. This finitization defines a bounded exploration space of size $4 \times 1 \times (4 \times 4)^3 = 16{,}384$. The Korat search generates the following candidate vectors for a SLL using this finitization:

0 0 0 0 0 0 0 0 :: 0
1 0 0 0 0 0 0 0 :: 0 2 3 1
1 1 0 0 0 0 0 0 :: 0 2 3 1
....
1 0 0 2 1 1 0 0 :: 0 2 3 4 5
1 0 0 2 1 2 0 0 :: 0 2 3 4 5
1 0 0 2 1 3 0 0 :: 0 2 3 4 5 6
1 0 0 2 1 3 1 0 :: 0 2 3 4 5 6 7 1 ∗∗∗
....

Each row shows two entities separated by `::`. The first entity on the left represents the candidate vector. The second entity on the right represents the field access order. Valid structures are annotated with `***`.

To illustrate, the candidate vector `[1 0 0 2 1 2 0 0]` represents an invalid SLL as shown in Fig. 2. This candidate vector shows that Node `1` is the header node. The next node of Node `1` is Node `2` and its value is `0`. The next node of Node `2` is Node `2` itself and its value is `1`. The next node of Node `3` is `null` and its value is `0`. Since Node `2` has a self-loop (cycle), it is an invalid SLL. Another example candidate vector `[1 0 0 2 1 3 1 0]` represents a valid SLL as shown in Fig. 3. This candidate vector shows that Node `1` is the header node. The next node of Node `1` is Node `2` and its value is `0`. Similarly, the next node of Node `2` is Node `3` and its value is `1`. The next node of Node `3` is `null` and its value is `1`. Since the SLL has no cycle and it has size `3` with `3` nodes reachable from the header, the SLL is valid.

2.2 Machine learning models

Ten machine learning models were used in this study, namely decision tree [51], ensemble tree classifiers, i.e., random forest tree [30], gradient boosting tree [23], Adaboost tree [22], support vector machine [10], multi-layer perceptron [48], k-nearest-neighbor [1], Gaussian naive Bayes [53], logistic regression classifier [2] and quadratic discriminant analysis [68].

```
public class SinglyLinkedList {
    Entry header;
    int size = 0;
    boolean repOK() {
        if (header == null)
            return false;
        if (header.element != null)
            return false;
        Set<Entry> visited = new java.util.HashSet<Entry>();
        visited.add(header);
        Entry current = header;
        while (true) {
            Entry next = current.next;
            if (next == null)
                break;
            if (next.element == null)
                return false;
            if (!visited.add(next))
                return false;
            current = next;
        }
        return visited.size() - 1 == size;
    }
    static IFinitization finSinglyLinkedList(int minSize, int maxSize, int numEntries, int numElems) {
        IFinitization f = FinitizationFactory.create(SinglyLinkedList.class);
        IObjSet entries = f.createObjSet(Entry.class);
        entries.setNullAllowed(true);
        entries.addClassDomain(f.createClassDomain(Entry.class, numEntries));
        IObjSet elems = f.createObjSet(SerializableObject.class);
        elems.setNullAllowed(true);
        elems.addClassDomain(f.createClassDomain(SerializableObject.class, numElems));
        IIntSet sizes = f.createIntSet(minSize, maxSize);
        f.set("header", entries);
        f.set("size", sizes);
        f.set("Entry.element", elems);
        f.set("Entry.next", entries);
        return f;
    }
}
```

Fig. 1 SinglyLinkedList Java class with *repOK* and finitization methods

2.2.1 Decision trees

Decision trees apply trees as classifiers. The leaf nodes represent the labels of the classes, and the inner nodes are a test on the feature. Decision trees are simple models that are easy to train and can be highly accurate in some cases. However, they are not good at modeling complex relationships and often suffer from over-fitting.

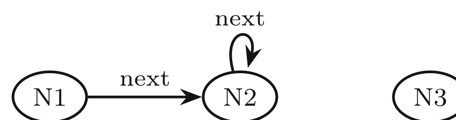

Fig. 2 An invalid SLL

Fig. 3 A valid SLL

2.2.2 Random forest tree

Random forest tree uses bagging (bootstrap aggregating) technique. This algorithm trains various decision trees with different characteristics which are then combined into one decision tree. Generally, the combined "forest" performs bet-

ter than decision tree, since they reduce variance without increasing bias. Furthermore, it reduces over-fitting which is one of the biggest problems for decision trees. Note that random forest uses highly efficient bagging techniques, which makes them highly usable and important in the field of ML.

2.2.3 Gradient boosting tree

A gradient boosting tree is a combination of numerous decision trees. A differentiable loss function is used, and the trees learn a set of parameters, which results in the lowest value for loss function.

2.2.4 Adaboost decision tree

An Adaboost tree is a type of ensemble tree. It is built on a set of decision trees, and this algorithm gradually learns from mistakes to improve itself over time. It analyzes which data samples were misclassified in each iteration. It then increases the weight of those data samples and tries to get them right in the next iterations. Data samples that are classified correctly are assigned a lower weight. Gradually, it separates the easier data samples from harder to classify data samples, from which the algorithm learns to do the classification.

2.2.5 Support vector machine

It is a non-probabilistic binary linear classifier. Data are mapped to a higher dimension which makes data samples linearly separable. This is called the *Kernel Trick* and is highly useful when the number of features is larger than available data samples for training. This model can work on a variety of kernels. However, it is sometimes difficult to choose the right kernel function, since finding a kernel metric together with the right hyper-parameters is not trivial.

2.2.6 Multi-layer perceptron

A multi-layer perceptron is one neural network kind. It consists of at least three layers where the first layer is known as the input layer and the last layer is known as the output layer. Layers between the first and the last layer are known as hidden layers. There can be multiple hidden layers with numerous neurons in each layer. They are often fully connected with each other. Each connection has a weight which is updated in each iteration using stochastic gradient descend [54]. As the number of hidden layers increases, the complexity of the model increases. And the accuracy increases up to a point where the model starts to over-fit and the accuracy starts to decrease. Finding the optimal number of hidden layers and neurons is often a difficult task. These models also need a large number of data samples for training. Therefore, they are often used to learn complex properties.

2.2.7 K-nearest neighbor

This algorithm is based on the principle of majority voting. Here, K means the number of nearest neighbors. For each data sample, it looks at the nearest k neighbors and chooses the majority class of the neighbors. Selecting the right value for k is important for the model's performance.

2.2.8 Gaussian naive Bayes

Gaussian naive Bayes assumes that the underlying distribution follows a Gaussian distribution. It is a simple probabilistic algorithm and has been widely used for text classification. This classifier only requires a small amount of training data which is why the training process is often fast for this model. However, when the conditional independence assumption of the Gaussian distribution is violated, this model may perform poorly.

2.2.9 Logistic regression classifier

This classifier uses a logistic function and assumes that the feature variables are dichotomous with no outliers in the dataset. It also assumes that there is little or no correlation between features. This algorithm estimates the logarithm of the odds which is a linear combination of features, and maximizes the likelihood function.

2.2.10 Quadratic discriminant analysis

Quadratic discriminant analysis creates a quadratic boundary between classes with numerous hyper-parameters. However, if there are many classes where the number of data samples is limited, this classifier may not perform well. In addition, this classification method assumes that the samples in each class follow a normal distribution. This would make it perform poorly if the assumption is violated, although it allows the covariance of each class to be different.

2.3 Data structure encoding

Korat represents data structures in a way that can be used as inputs for machine learning models without a lot of preprocessing. For a Korat state vector, we simply use it as the feature vector of the ML models. Each element in the vector has a value between 0 and the number of the possible values of that element. We mark valid data structures as positive samples (label 1) and invalid data structures as negative samples (label 0).

3 Experimental evaluation

This section explains our study methodology and evaluation results on the ten data structure properties using ten ML models. The ML models include decision tree (DT) [51], ensemble tree classifiers, i.e., random forest tree (RFT) [30], gradient boosting tree (GBT) [23], Adaboost tree (ABT) [22], support vector machine (SVM) [10], multi-layer perceptron (MLP) [48], k-nearest neighbor (KNN) [1], Gaussian naive Bayes (GNB) [53], logistic regression classifier (LRC) [2] and quadratic discriminant analysis (QDA) [68].

We used ten standard data structure subjects from the publicly available Korat distribution [37], namely binary heap (BH), binary search tree (BST), binary tree (BT), directed acyclic graph (DAG), disjoint set (DS), Fibonacci heap (FH), heap array (HA), red black tree (RBT), sorted list (SL) and singly linked list (SLL). Prior work used similar subjects for their evaluation [7,15–17,46,60].

Execution platform All experiments were performed on a machine with a 4-core 2.20 GHz Intel processor and 16GB of RAM, running Ubuntu 16.04 LTS.

3.1 Generation of datasets

For each data structure subject, we used Korat [4] to generate four datasets denoted by α, β, γ and δ. Each dataset contains positive samples that represent the cases when a data structure property holds and negative samples that represent the cases where the data structure property does not hold. For each data structure, Korat generates feature vectors in a format such that each bit represents a node or an element. Korat also generates ground truth labels for each data sample. The size of the feature vector was kept constant to make it compatible with existing ML tools. In the future, we plan to study the impact of feature vectors with variable length on the learnability of ML models.

For each data structure subject, more negative data samples were generated as compared to positive data samples. This is expected since the number of invalid data structures is much higher than valid data structures. Without sampling datasets, we will bias our models to always predict negative class and still achieve high accuracy. Thus, we used the undersampling technique [50] so that we have the same number of positive and negative samples in our dataset. We kept all of the positive data samples and randomly selected the same number of negative samples. Besides, we do not have any duplicated samples in our datasets. For dataset α, the size (ϕ) of each data structure subject was chosen such that Korat generates at least 10,000 positive samples. For dataset β, size from 1 up to ϕ was chosen so that a comparison between the two types of datasets can be made. The only difference between α and β datasets is the size of the data structures. A fixed size for each data structure was used to generate dataset α, whereas a varied size (i.e., up to a certain size) was used for generating β dataset. For example, a size of 10 was used to generate α for binary search tree. However, β consists of binary search trees from size 1 up to 10. Datasets γ and δ are like α and β, respectively, but with one-hot encoding format [28].

One-hot encoding format For one-hot encoding, we represent each element of the original candidate vector using a binary vector of length θ (number of possible values of that element). For example, the SLL shown in Fig. 3 has a candidate vector of length 8. The value of each index represents a node range from `0` to `3`, representing four possibilities: [`null`, Node `1`, Node `2` and Node `3`]. Index 1 (size of the list) can have one value. Therefore, we create a candidate vector consisting of 29 bits (4 bits to represent each of the seven elements and 1 bit to represent size of the list). `null` is represented by `[1 0 0 0]`, Node `1` is represented by `[0 1 0 0]`, Node `2` is represented by `[0 0 1 0]`, and Node `3` is represented by `[0 0 0 1]`.

Table 1 shows the size for each of the data structures in dataset α and γ. It also lists the total state space, the number of valid structures explored, the number of invalid structures explored and the total number of structures explored by Korat. Table 2 shows the size for each of the data structure in dataset β and δ. It also lists the total state space, the number of valid structures explored, the number of invalid structures explored and the total number of structures explored by Korat.

3.2 Training ML models

Previously, we used four different train–test ratios to study the effect of different ratios on the learnability of ML models. This study extends previous work by going to an extreme ratio of 1:99. We want to investigate how the model would perform under the extreme circumstances where it is trained on just 1% of the dataset and tested on the rest of 99%. To our knowledge, this study is first to evaluate the learnability of ML models with such a small percentage of dataset kept for training. We also explored hyper-parameter tuning. We did find that performance metrics improve. However, the small increase in performance metrics is offset by a much larger time taken by ML models to train. Therefore, this study reports the results of ML models using default settings. In addition, Scikit-Learn library [57] was used for training and testing ML models. Table 3 shows the parameters used during the training of each ML model. Figure 4 gives an overview of our experimental setup.

3.3 Performance metrics

We report the counts of true positives (TP), false negatives (FN), false positives (FP) and true negatives (TN). If a data

Table 1 Candidate structures explored by Korat for each data structure subject (datasets α and γ)

Subject	Size	State space	Valid explored	Invalid explored	Total explored
BH	7	2^{109}	107,416	154,372	261,788
BST	10	2^{105}	16,796	155,439,076	155,455,872
BT	10	2^{72}	16,796	798,304	815,100
DAG	6	2^{108}	19,696	185,034	204,730
DS	5	2^{39}	41,546	372,309	413,855
FH	5	2^{82}	52,281	112,084	164,365
HA	6	2^{23}	13,139	51,394	64,533
RBT	10	2^{193}	17,160	32,074,894	32,092,054
SL	8	2^{93}	12,870	90,258	103,128
SLL	9	2^{72}	21,147	268,177	289,324

Table 2 Candidate structures explored by Korat for each data structure subject (datasets β and δ)

Subject	Size	State space	Valid explored	Invalid explored	Total explored
BST	10	2^{109}	223,191	216,457,718	216,680,909
BT	10	2^{76}	23,714	1,028,526	1,052,240
RBT	10	2^{196}	18,439	10,925,390	10,943,829
SL	8	2^{96}	24,310	150,962	175,272
SLL	9	2^{75}	26,443	500,868	527,311

sample is a valid structure and classifier correctly predicts it to be a valid structure, we count it as a TP. If a data sample is a valid structure and classifier incorrectly predicts it to be an invalid structure, we count it as a FN. If a data sample is an invalid structure and classifier incorrectly predicts it to be a valid structure, we count it as a FP. If a data sample is an invalid structure and classifier correctly predicts it to be an invalid structure, we count it as a TN. We use the traditional metrics of accuracy (Acc), precision (Prec), recall (Recall) and F1-score (F1). Accuracy is calculated as $\frac{TP+TN}{TP+TN+FP+FN}$. Precision is calculated as $\frac{TP}{TP+FP}$. Recall is calculated as $\frac{TP}{TP+FN}$. F1-score is calculated as $\frac{2*Precision*Recall}{Precision+Recall}$.

3.4 Research questions

We answer the following research questions in our study.

- **RQ1:** Are certain ML models more suitable for learning data structure properties?
- **RQ2:** How do ML models perform when trained with different train–test ratios?
- **RQ3:** Are certain data structure properties more learnable than others?
- **RQ4:** Does the learnability of varied-size data structures differ from the learnability of fixed-size data structures?
- **RQ5:** Does one-hot encoding representation impact the learnability of data structure properties?

We report the average performance metrics in this section. The train/test datasets, as well as detailed results and counts of TP, FP, FN and TN, are available at https://github.com/muhammadusman93/STTTKorat.

RQ1: Are certain ML models more suitable for learning data structure properties?
Table 4 shows the performance of ML models for each dataset. Figure 5 summarizes the accuracy of each ML model. We compare accuracy (across all four datasets) to see whether certain ML models are more suitable for learning data structure properties. For α dataset, DT performs slightly better than MLP. However, for β,γ and δ datasets, MLP performs the best. This might be because MLP does not make any assumptions regarding the underlying probability density functions or distributions of the data. MLP can learn nonlinear complex functions and generalize well beyond training data. The median accuracy for GBT and ABT (0.98 and 0.96, respectively) is slightly lower than median accuracy of DT (0.99). However, on average ensemble trees, i.e., ABT, GBT and RFT, perform better than DT because ensemble trees are combinations of different decision trees and have better prediction ability. SVM performs well with median accuracy around 0.96 because it can learn complex problems using kernel trick and scale well to high-dimensional data. LRC is one of the best models (median accuracy of 0.95) because LRC is less prone to over-fitting and performs extremely well if the dataset is linearly separable. KNN also performs well. GNB performs poorly (median accuracy of 0.71) because it assumes that the underlying distribution follows a Gaus-

Table 3 Parameters for training ML models (Scikit-Learn library [57])

Models	Parameter	Description	Value
DT	criterion	Criteria to measure quality of a split	Gini
	max_features	Number of features considered for the best split	n_features
	min_impurity_split	Threshold for early stopping in tree growth	0.0000001
	min_samples_leaf	Minimum samples required to be at a leaf node	1
	min_samples_split	Minimum samples required to split an internal node	2
	splitter	Select the best strategy to choose the split at each node	Best
RFT	criterion	Criteria to measure quality of a split	Gini
	max_features	Number of features considered for the best split	sqrt(n_features)
	min_impurity_split	Threshold for early stopping in tree growth	0.0000001
	min_samples_leaf	Minimum samples required to be at a leaf node	1
	min_samples_split	Minimum samples required to split an internal node	2
	n_estimators	Number of trees in the forest	100
GBT	criterion	Function to measure the quality of a split	Friedman_mse
	learning_rate	Learning rate	0.1
	max_depth	Maximum depth of individual regression estimators	3
	max_features	Number of features considered for the best split	n_features
	min_impurity_split	Threshold for early stopping	0.0000001
	min_samples_leaf	Minimum samples required to be at a leaf node	1
	min_samples_split	Minimum samples required to split an internal node	2
	n_estimators	Number of boosting stages	100
	subsample	Fraction of samples used for fitting individual base learner	1.0
	tol	Tolerance for early stopping	0.0001
ABT	algorithm	Algorithm	SAMME.R
	learning_rate	Learning rate	1
	n_estimators	Maximum number of estimators at which boosting is terminated	50
SVM	*C*	Regularization parameter	1.0
	kernal	Kernel function	rbf
	max_iter	Limit on the number of iterations (− 1 means no limit)	− 1
	shrinking	Whether to use shrinking	True
	tol	Tolerance for stopping criterion	0.001
MLP	activation	Activation function	Relu
	alpha	L2 regularization penalty	0.0001
	batch_size	Batch size	min(200, n_samples)
	beta_1	Exponential decay rate for estimates of first moment vector in Adam	0.9
	beta_2	Exponential decay rate for estimates of second moment vector in Adam	0.999
	epsilon	Numerical stability in Adam	0.00000001
	hidden_layer_sizes	Number of units in each hidden layer, Number of hidden layers (1)	(100)
	learning_rate_init	Initial learning rate	0.001
	max_iter	Number of iterations	200
	n_iter_no_change	Epochs after which training is stopped (if there is no improvement in performance)	10

Table 3 continued

Models	Parameter	Description	Value
	solver	Solver	Adam
	tol	Tolerance	0.0001
KNN	algorithm	Framework can choose the best algorithm	auto
	n_neighbors	Number of neighbors to use for k-neighbors queries	5
	p	Power parameter for the Minkowski metric (2 means Euclidean_distance)	2
	weights	Set equal weights to all neighborhood points	uniform
GNB	var_smoothing	Portion of the largest variance of all features (added to variances to improve calculation stability)	0.000000001
LRC	C	Inverse of regularization strength	1.0
	fit_intercept	Bias added to the decision function	true
	max_iter	Maximum number of iterations taken by solver to converge	100
	penalty	Regularization	l2
	solver	Solver	lbfgs
	tolfloat	Tolerance value	0.0001
QDA	tol	Threshold for rank estimation	0.0001

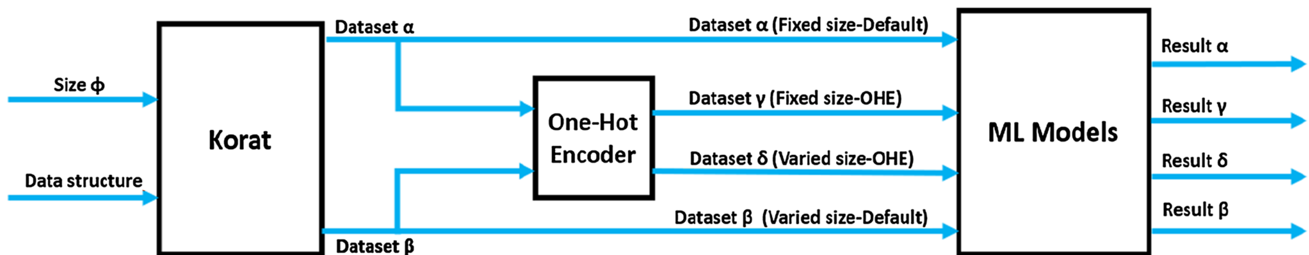

Fig. 4 Architecture of the experimental setup

Table 4 Average performance across all ten data structure subjects and five train–test ratios for each ML model

Models	Acc α	Prec α	Recall α	F1 α	Acc β	Prec β	Recall β	F1 β	Acc γ	Prec γ	Recall γ	F1 γ	Acc δ	Prec δ	Recall δ	F1 δ
DT	**0.98**	**0.97**	**0.98**	**0.98**	**0.99**	**0.99**	0.99	**0.99**	0.97	0.96	0.97	0.97	0.86	0.86	0.84	0.85
RFT	0.96	0.96	0.96	0.96	**0.99**	0.98	0.99	**0.99**	0.95	0.95	0.96	0.95	0.87	0.86	0.88	0.87
GBT	0.96	0.95	0.97	0.96	0.98	0.97	0.99	0.98	0.95	0.93	0.97	0.95	0.90	0.86	0.99	0.91
ABT	0.90	0.90	0.91	0.90	0.94	0.93	0.96	0.94	0.90	0.89	0.91	0.90	0.87	0.83	0.96	0.88
SVM	0.91	0.90	0.91	0.90	0.95	0.97	0.94	0.94	0.91	0.89	0.93	0.90	0.85	0.78	0.88	0.83
MLP	0.96	0.96	0.97	0.96	**0.99**	0.98	**1.00**	**0.99**	**0.98**	**0.97**	**0.98**	**0.98**	**0.91**	**0.88**	0.98	**0.93**
KNN	0.88	0.85	0.95	0.89	0.96	0.93	0.99	0.96	0.84	0.81	0.95	0.87	0.78	0.76	0.89	0.81
GNB	0.70	0.66	0.90	0.74	0.76	0.74	0.93	0.79	0.71	0.68	0.90	0.75	0.68	0.63	**1.00**	0.76
LRC	0.87	0.86	0.90	0.88	0.95	0.93	0.97	0.95	0.90	0.89	0.91	0.90	0.87	0.83	0.95	0.88
QDA	0.66	0.56	0.62	0.54	0.60	0.39	0.58	0.46	0.74	0.73	0.84	0.73	0.70	0.64	0.99	0.77

Bold values highlight the highest value for each column

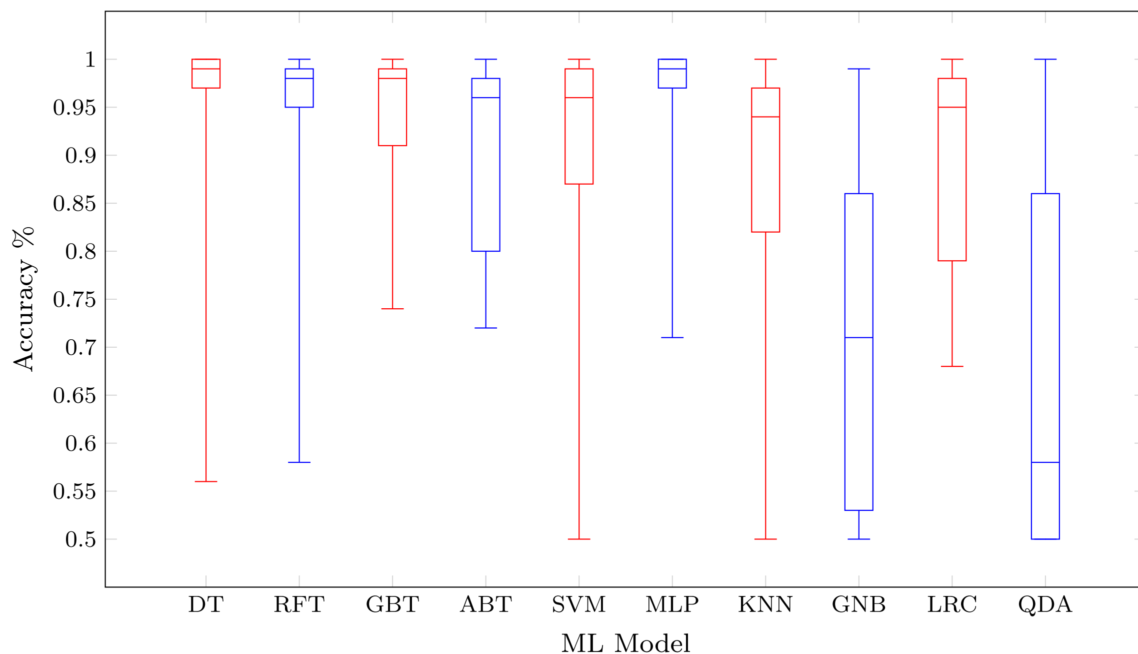

Fig. 5 Accuracy across all ten data structure subjects, five train–test ratios and four datasets for each ML model

Table 5 Average performance across all ten data structure subjects and ten ML models for each train–test ratio

Ratio	Acc α	Prec α	Recall α	F1 α	Acc β	Prec β	Recall β	F1 β	Acc γ	Prec γ	Recall γ	F1 γ	Acc δ	Prec δ	Recall δ	F1 δ
75:25	**0.89**	**0.87**	**0.93**	**0.89**	0.91	**0.89**	0.93	0.89	**0.90**	**0.89**	**0.96**	**0.92**	**0.84**	**0.80**	**0.96**	0.86
50:50	**0.89**	0.86	**0.93**	**0.89**	**0.92**	0.88	**0.94**	**0.91**	**0.90**	0.88	0.95	0.91	**0.84**	**0.80**	**0.96**	**0.87**
25:75	**0.89**	0.86	0.92	0.88	0.90	0.87	0.93	0.89	**0.90**	0.88	0.95	0.91	0.82	0.79	0.94	0.85
10:90	0.88	0.85	0.91	0.87	**0.92**	0.88	0.93	0.90	0.89	0.87	0.94	0.90	0.83	**0.80**	0.95	0.86
1:99	0.85	0.83	0.85	0.82	0.89	0.87	0.91	0.88	0.84	0.83	0.85	0.82	0.80	0.76	0.86	0.80

Bold values highlight the highest value for each column

sian distribution and our datasets do not necessarily have a Gaussian distribution. Similarly, QDA performs the worst (median accuracy of 0.58) because it works well only when the datasets follow a normal distribution, while it is not the case for our datasets.

In summary, we can conclude that MLP, RFT, ABT, GBT, DT, SVM, KNN and LRC are good models to learn data structure properties. GNB and QDA performed the worst, and study shows that they are not good for learning data structure properties. Thus, we can say that *certain ML models are more suitable for learning data structure properties*. And this gives us good indications of which ML models to select in learning data structure properties.

RQ2: How do ML models perform when trained with different train–test ratios?

Table 5 summarizes the performance of ML models for different train–test ratios. We compare how decreasing the train–test ratio (from 75:25 to 1:99) impacts the learnability of ML models. For α dataset, accuracy decreases by 4% and F1-score decreases by 7%. For β dataset, accuracy decreases by 2% and F1-score decreases by 1%. For γ dataset, accuracy decreases by 6% and F1-score decreases by 10%. For δ dataset, accuracy decreases by 4% and F1-score decreases by 6%. The decrease in accuracy is in the range of [2%, 6%] and a decrease in F1-score is in the range of [1%, 10%]. This shows that the performance of the ML models decreases, as the train–test ratio decreases. This is expected because when the machine learning model is trained on a smaller dataset, it may not get all properties for a data structure. Only a limited number of rules learned would result in bad prediction performance. However, it is surprising to see that even with just 1% of the data used for training, accuracy and F1-score are always above 0.79. This gives hints that data structure properties can be learned using a much smaller dataset, which would allow us to learn data structures of much bigger sizes. To summarize, these experiments show that *ML models are very good in learning data structure properties even with a very small portion of data for training*.

RQ3: Are certain data structure properties more learnable than others?

Table 6 Average performance across all ten ML models and five train–test ratios for each data structure subject

Subjects	Acc α	Prec α	Recall α	F1 α	Acc β	Prec β	Recall β	F1 β	Acc γ	Prec γ	Recall γ	F1 γ	Acc δ	Prec δ	Recall δ	F1 δ
BH	**0.98**	**0.97**	0.99	**0.98**	–	–	–	–	**0.98**	**0.97**	0.99	**0.98**	–	–	–	–
BST	0.91	0.92	0.95	0.91	0.90	0.88	0.98	0.92	0.92	0.92	0.97	0.93	0.88	0.86	0.99	0.91
BT	0.80	0.73	0.82	0.76	0.91	0.86	0.87	0.86	0.87	0.85	0.93	0.88	0.77	0.71	0.87	0.78
DAG	0.81	0.80	0.88	0.83	–	–	–	–	0.79	0.77	0.88	0.81	–	–	–	–
DS	0.80	0.77	0.76	0.74	–	–	–	–	0.81	0.80	0.78	0.76	–	–	–	–
FH	0.83	0.81	0.93	0.86	–	–	–	–	0.82	0.81	0.92	0.85	–	–	–	–
HA	0.84	0.82	0.87	0.83	–	–	–	–	0.84	0.82	0.87	0.83	–	–	–	–
RBT	0.97	0.96	0.98	0.97	**0.95**	**0.93**	**0.99**	**0.96**	0.96	0.94	**1.00**	0.96	**0.94**	**0.91**	**1.00**	**0.95**
SL	0.90	0.90	**1.00**	0.93	0.87	0.86	0.96	0.89	0.91	0.90	**1.00**	0.93	0.86	0.85	0.96	0.89
SLL	0.94	0.89	0.90	0.89	0.92	0.87	0.88	0.87	0.95	0.94	0.99	0.96	0.70	0.63	0.86	0.72

The ”–” indicates that Korat cannot generate a varied-size data structure for a given subject
Bold values highlight the highest value for each column

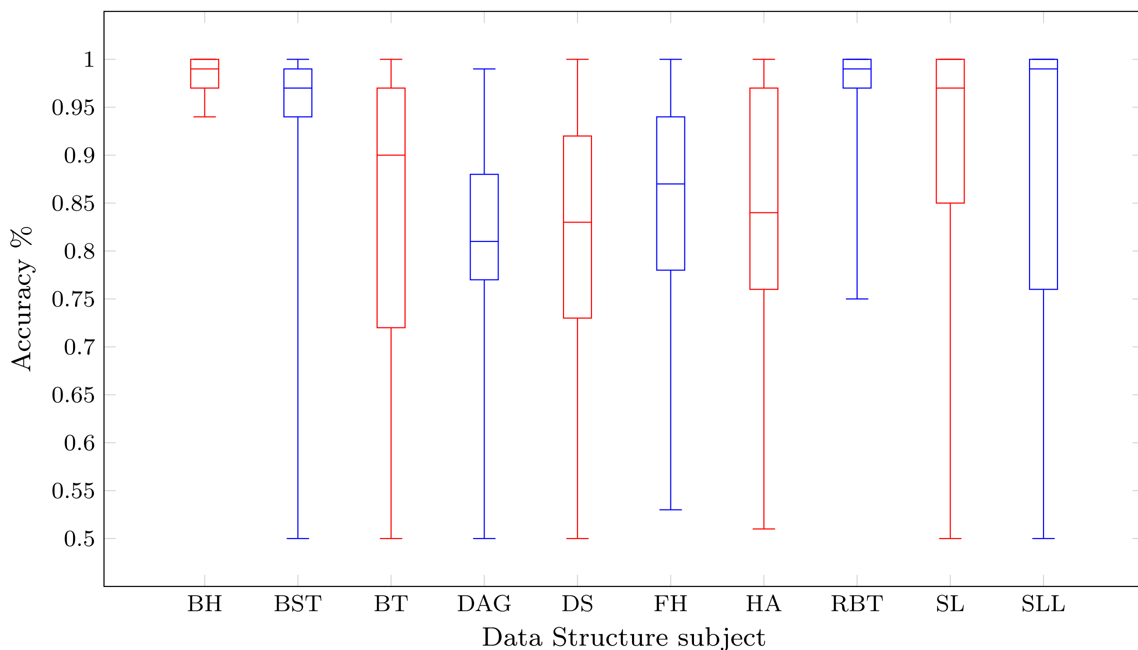

Fig. 6 Accuracy across all ten ML models, five train–test ratios and four datasets for each data structure subject

Table 6 summarizes the performance of ML models for each data structure, for each dataset. Figure 6 summarizes the accuracy of each data structure subject. BST performs better (median accuracy of 0.97) than BT (median accuracy of 0.90). On the one hand, BT might be more learnable since the ML model only needs to learn 1 property, i.e., acyclicity for BT, whereas the model needs to learn two properties, i.e., correct order property and acyclic property for BST. On the other hand, learning that the nodes are in the correct order is much easier than to learn that a tree is acyclic. Results show that an ML model can differentiate lots of invalid BST just based on the correct order property. SL is more learnable (median accuracy of 0.97) than DAG (median accuracy of 0.81). Since it is easier for an ML model to recognize a sorted list (bits in feature vector are in ascending order) than to check whether a graph is acyclic. Overall, complex data structures (BH, RBT, BST and SL) are more learnable ones. We investigated that complex data structures have multiple properties which makes classification even easier, while simple structures (DAG and BT) give only one or two properties which makes the classification more difficult.

To summarize, BH and RBT are the most learnable properties. Study shows that DAG is harder to learn as compared to other data structure properties. Thus, we can say that *certain data structure properties are more learnable than other data structure properties*.

RQ4: Does the learnability of varied-size data structures differ from the learnability of fixed-size data structures?

Table 6 shows the performance of ML models for fixed-size data structures (α) and varied-size data structures (β). Korat only supports five varied-size data structures including BST, BT, RBT, SL and SLL. We compare the change in performance metrics when fixed-size data structures are replaced by the corresponding varied-size data structures. Accuracy increases by 11% for BT; decreases by 1% for BST; decreases by 3% for SL; and decreases by 2% for RBT and SLL. We can see that accuracy increases only in one subject and it decreases in four subjects. (Maximum drop in accuracy is 3%.) This shows no significant differences between the two types of data structures.

This is expected because ML models learn properties specific to the data structure subject regardless of its size. For example, for binary search tree, it learns that there should be no cycle in the tree and all the nodes should be in the correct order and be reachable. However, it is not the size of the tree that the model learns about. To summarize, experimental results show that *there is no significant learnability difference between varied-size and fixed-size data structures*.

RQ5: Does one-hot encoding representation impact the learnability of data structure properties?

Table 6 summarizes the performance of ML models for each encoding format, i.e., Korat's default encoding (α and β) with one-hot encoding (γ and δ). We compare Korat's default encoding with one-hot encoding and see how the one-hot encoding representation impacts the learnability of ML models for both varied-size and fixed-size data structures.

Varied-size data structures Accuracy decreases by 2% for BST; decreases by 1% for RBT and SL; decreases by 14% for BT; and decreases by 22% for SLL. The decrease in accuracy is in the range of [1%, 22%]. These results show that there is a significant decrease in performance if the one-hot encoding is applied. This is expected since one-hot encoding increases the number of features which ML models would find it difficult to learn. To summarize, we can conclude that *using one-hot encoding representation reduces the learnability of ML models for varied-size data structures*.

Fixed-size data structures Accuracy increases by 1% for BST, SL, SLL and DS; increases by 7% for *BT*; decreases by 1% for RBT and FH; decreases by 2% for DAG; and stays the same for BH and HA. The accuracy increases in five out of these ten fixed-size subjects. The change in accuracy is in the range of [−2%, 7%]. These results show that there would be a significant increase in performance if the one-hot encoding is applied. These results are surprising as we expected that one-hot encoding should decrease the performance of ML models. One reason can be that linear classifiers like LRC and GNB perform better if the distance between data samples is high. This is indeed the case with one-hot encoding format. We believe that the reason for this result deserves further investigation and analysis. To summarize, we could say that *using one-hot encoding representation improves the learnability of ML models for fixed-size data structures*.

Applications Our objective is to perform controlled experiments to understand the potential role of ML methods in learning data structure properties. These properties, once learned, can be used to provide a run-time check to see whether a program state at a particular point conforms to the learned property. A trained classifier can also be used when *repOK* is not available. These learned classifiers can also provide checks to enable automated theorem proving [12], error recovery [13,36], static analysis [24,58] and automated test input generation [4,38]. Test generation tools can generate data structures and simply use trained ML models to verify whether these structures are valid. We believe that verification via trained ML models would be efficient compared to executing numerous checks on each data structure instance. Thus, the use of ML models in software analysis holds a promising future and we expect that new techniques would emerge to apply ML in analyzing and finding bugs in software systems.

4 Threats to validity

In our experiments, we selected the scope such that at least 10,000 positive data samples were generated. However, the machine learning models may perform differently if trained and tested on scopes that generate millions of data samples. We also used ten data structure subjects for evaluation. It is highly important to extend this study on graphs and more complex data structures like AVL trees. We also balanced the proportion of positive and negative samples. It would be interesting to see how the model performs when no balancing technique is applied. The models may be biased in predicting all samples as negative. It will be interesting to see how machine learning models can be trained on data structures of smaller sizes and then used to classify data structures of bigger sizes. For example, we can train our models on binary trees of size exactly 6 and then classify binary trees of size exactly 10. Korat is a state-of-art tool used to generate data structures and it always annotates valid and invalid data structures correctly, which means there is no noise in our datasets. We may add some random noise to our datasets to study the impact, which we leave as our future work.

5 Related work

Use of ML in software engineering is burgeoning [5,8,21, 24,41,59]. It is used in many applications, including automatic program repair and bug detection. ML is also used to learn graphs and relational properties [35]. Numerous frameworks such as Vapnik–Chervonenkis [66] (VC dimension)

and probably approximately correct (PAC) [65] have been used to study learnability of ML models. Lot of work has been done in the field of invariant generation [31] including sketching [62] and program synthesis [3,27,42]. ML models are believed to guide test generation programs in their search [33,61]. There is also work in the field of static and dynamic analysis [14,19,34,43,47,52,55,56,69].

Daikon [19,20] is a state-of-the-art technique used for the invariant generation. It uses a collection of property templates, observes the program states and checks whether the properties are satisfied. However, the technique has limited applicability for structural properties. Derayft [40] extends the work of Daikon and can efficiently work with data structures, although it requires a collection of property templates which is often limited.

Malik et al. [41] used a support vector machine for characterizing the program properties. While it requires the use of complex graph spectra [6] techniques, their study was confined to just one ML model, which may not be generalized to other ML models. More recent work [21] studied the use of neural networks in learning data structures properties. They used Randoop [49] for test generation and showed that ML models perform better than Daikon [19] in some cases. However, only neural networks and six data structure properties were studied. In contrast, our study involves ten data structure subjects and ten ML models with five different train–test ratios. We also compare the learnability of varied-size data structures with the learnability of fixed-size data structures. Usman et al. recently introduced [63,64] a new approach to quantify quality of trained models based on model counting [26] and applied it in the context of properties of binary relations. We plan to employ model counting in future work to build on our study in this paper.

6 Conclusion

This paper presented a systematic study of the learnability of data structure properties. The study shows that while most ML models are highly accurate in learning data structure properties, some ML models such as *quadratic discriminant analysis* and *Gaussian naive Bayes* are not suitable for learning these properties. It also found that decreasing train–test ratio from 75:25 to 1:99 reduces accuracy by at most 6%, which opens the possibility of learning data structure properties of larger sizes. Another finding is that certain data structure properties are more learnable than others, e.g., *binary heap* and *red black tree*. Our study also compares the learnability of varied-size data structures with the learnability of fixed-size data structures. We did not find any significant difference in the learnability of these two types of data structures. We also investigate the performance of ML models trained on different encodings: one-hot encoding and Korat's default encoding. Results show that one-hot encoding representation hinders the performance of varied-size data structures but improves the performance of fixed-size data structures. Overall, these results show that the use of ML models for learning data structure properties is highly promising.

Acknowledgements We thank Rohan Garg, Emily Ginsburg, Michael Herrington, Tara Kuruvilla, Raghav Prakash and the anonymous reviewers for helpful feedback and comments. This research was partially supported by the US National Science Foundation under Grant Nos. CCF-1704790 and CCF-1718903.

References

1. Altman, N.S.: An introduction to kernel and nearest-neighbor nonparametric regression. Am. Stat. **46**(3), 175–185 (1992)
2. Bacaër, N.: Verhulst and the logistic equation **01**, 1838 (2011)
3. Bodik, R.: Program synthesis: opportunities for the next decade. In: International Conference on Functional Programming, pp. 1–1 (2015)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: International Symposium on Software Testing and Analysis, pp. 123–133 (2002)
5. Briand, L.C., Labiche, Y., Liu, X.: Using machine learning to support debugging with tarantula. In: International Symposium on Software Reliability, pp. 137–146 (2007)
6. Brouwer, A.E., Haemers, W.H.: Spectra of Graphs. Springer, New York (2012)
7. Çelik, A., Pai, S., Khurshid, S., Gligoric, M.: Bounded exhaustive test-input generation on GPUs. PACMPL **1**(OOPSLA), 94:1–94:25 (2017)
8. Chen, Y.-F., Hong, C.-D., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Formal Methods in Computer Aided Design, pp. 76–83 (2017)
9. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: Design Automation Conference, pp. 368–371 (2003)
10. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
11. Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: Dynamic symbolic execution for invariant inference. In: International Conference on Software Engineering, pp. 281–290 (2008)
12. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: International Conference on Automated Deduction, pp. 378–388 (2015)
13. Demsky, B., Rinard, M.C.: Automatic detection and repair of errors in data structures. In: Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 78–95 (2003)
14. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: International Conference on Object Oriented Programming Systems Languages and Applications, pp. 443–456 (2013)
15. Dini, N., Yelen, C., Alrmaih, Z., Kulkarni, A., Khurshid, S.: Korat-API: a framework to enhance korat to better support testing and reliability techniques. In: International Symposium on Applied Computing, pp. 1934–1943 (2018)
16. Dini, N., Yelen, C., Gligoric, M., Khurshid, S.: Extension-aware automated testing based on imperative predicates. In: Conference on Software Testing, Validation and Verification, pp. 25–36 (2019)
17. Dini, N., Yelen, C., Khurshid, S.: Optimizing parallel Korat using invalid ranges. In: International Symposium on Model Checking of Software, pp. 182–191 (2017)

18. Elkarablieh, B., Garcia, I., Suen, Y.L., Sarfraz, K.: Assertion-based repair of complex data structures. In: International Conference on Automated Software Engineering, pp. 64–73 (2007)
19. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: International Conference on Software Engineering, pp. 449–458 (2000)
20. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
21. Facundo, M., Degiovanni, R., Ponzio, P., Regis, G., Aguirre, N., Frias, M.F.: Training binary classifiers as data structure invariants. In: International Conference on Software Engineering, pp. 759–770 (2019)
22. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. Syst. Sci. **55**(1), 119–139 (1997)
23. Friedman, J.H.: Greedy function approximation: A gradient boosting machine. Ann. Statist. **29**(5), 1189–1232 (2001)
24. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Symposium on Principles of Programming Languages, pp. 499–512 (2016)
25. Godefroid, P.: Model checking for programming languages using verisoft. In: Symposium on Principles of Programming Languages, pp. 174–186 (1997)
26. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting (2008)
27. Gulwani, S.: Dimensions in program synthesis. In: International Symposium on Principles and Practice of Declarative Programming, pp. 13–24 (2010)
28. Guo, C., Berkhahn, F.: Entity embeddings of categorical variables. CoRR (2016). arXiv:1604.06737
29. Hernandez, J., Carrasco-Ochoa, J.A., Martínez-Trinidad, J.F.: An empirical study of oversampling and undersampling for instance selection methods on imbalance datasets. In: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, pp. 262–269. Springer (2013)
30. Ho, T.K.: Random decision forests. In: International Conference on Document Analysis and Recognition (1995)
31. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 60–64. Springer (2011)
32. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: International Symposium on Software Testing and Analysis, pp. 14–25 (2000)
33. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: International Conference on Software Engineering, pp. 215–224 (2010)
34. Jump, M., McKinley, K.S.: Dynamic shape analysis via degree metrics. In: International Symposium on Memory Management, pp. 119–128 (2009)
35. Kazemi, S.M., Poole, D.: Relnn: A deep neural model for relational learning (2017)
36. Ke, Y., Stolee, K.T, Goues, C.L., Brun, Y.: Repairing programs with semantic code search (T). In: International Conference on Automated Software Engineering, pp. 295–306 (2015)
37. Korat GitHub repository. https://github.com/korattest/korat
38. Korel, B.: Automated software test data generation. Trans. Softw. Eng. **16**(8), 870–879 (1990)
39. Liskov, B., Guttag, J.V.: Program Development in Java-Abstraction, Specification, and Object-Oriented Design. Addison-Wesley, Boston (2001)
40. Malik, M., Pervaiz, A., Uzuncaova, E., Khurshid, S.: Deryaft: A tool for generating representation invariants of structurally complex data. In: International Conference on Software Engineering, pp. 859–862 (2008)
41. Malik, M.Z.: Dynamic shape analysis of program heap using graph spectra: NIER track. In: International Conference on Software Engineering, pp. 952–955 (2011)
42. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
43. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 413–427 (2008)
44. Mera, E., Lopez-García, P., Hermenegildo, M.: Integrating software testing and run-time checking in an assertion verification framework. In: Logic Programming, pp. 281–295. Springer (2009)
45. Meyer, B.: Class invariants: concepts, problems, solutions. CoRR (2016). arXiv:1608.07637
46. Misailovic, S., Milicevic, A., Petrovic, N., Khurshid, S., Marinov, D.: Parallel test generation and execution with Korat. In: Symposium on the Foundations of Software Engineering, pp. 135–144 (2007)
47. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Conference on Programming Language Design and Implementation, pp. 221–231 (2001)
48. Murtagh, F.: Multilayer perceptrons for classification and regression. Neurocomputing **2**(5), 183–197 (1991)
49. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: International Conference on Software Engineering, pp. 75–84 (2007)
50. Provost, F.: Machine learning from imbalanced data sets 101. In: Proceedings of the AAAI Workshop on Imbalanced Data Sets, pp. 1–3 (2000)
51. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
52. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Symposium on Logic in Computer Science, pp. 55–74 (2002)
53. Rish, I.: An empirical study of the naive bayes classifier. In: IJCAI, pp. 3 (2001)
54. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat. **22**(3), 400–407 (1951)
55. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Symposium on Principles of Programming Languages, pp. 105–118 (1999)
56. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: Symposium on Principles of Programming Languages, pp. 318–329 (2004)
57. Scikit-Learn Library. https://scikit-learn.org/stable/. Accessed 18 Apr 2019
58. Si, X., Dai, H., Raghothaman, M., Naik, M., Le, S.: Learning loop invariants for program verification. In: Conference on Neural Information Processing Systems, pp. 7762–7773 (2018)
59. Si, X., Dai, H., Raghothaman, M., Naik, M., Le, S.: Learning loop invariants for program verification. In: Advances in Neural Information Processing Systems, pp. 7751–7762 (2018)
60. Siddiqui, J.H., Khurshid, S.: PKorat: Parallel generation of structurally complex test inputs. In: International Conference on Software Testing Verification and Validation, pp. 250–259 (2009)
61. Singh, S., Zhang, M., Khurshid, S.: Learning guided enumerative synthesis for superoptimization. In: International Symposium on Model Checking of Software, p. 172–192 (2019)
62. Solar-Lezama, A.: Program Synthesis by Sketching. PhD thesis (2008)
63. Usman, M., Wang, W., Vasic, M., Wang, K., Vikalo, H., Khurshid, S.: A study of the learnability of relational properties. In: 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). To appear(2020)
64. Usman, M., Wang, W., Wang, K., Yelen, C., Dini, N., Khurshid, S.: A study of learning data structure invariants using off-the-shelf

tools. In: International Symposium on Model Checking of Software, pp. 226–243 (2019)
65. Valiant, L.G.: A theory of the learnable. CACM **27**(11) (1984)
66. Vapnik, V.N., Chervonenkis, A.Ya.: On the uniform convergence of relative frequencies of events to their probabilities. In: Measures of Complexity: Festschrift for Alexey Chervonenkis. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21852-6_3
67. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: International Conference on Automated Software Engineering, pp. 3–12 (2000)
68. Wu, W., Mallet, Y., Walczak, B., Penninckx, W., Massart, D.L., Heuerding, S., Erni, F.: Comparison of regularized discriminant analysis linear discriminant analysis and quadratic discriminant analysis applied to nir data. Anal. Chim. Acta **329**(3), 257–265 (1996)
69. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: Conference on Programming Language Design and Implementation, pp. 349–361 (2008)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.