A Study of Learning Data Structure Invariants Using Off-the-shelf Tools

Muhammad Usman(✉), Wenxi Wang(✉), Kaiyuan Wang(✉), Cagdas Yelen(✉), Nima Dini(✉), and Sarfraz Khurshid(✉)

University of Texas at Austin, Austin, TX 78712, USA
{muhammadusman,wenxiw,kaiyuanw,cagdas,nima.dini,khurshid}@utexas.edu

Abstract. Data structure invariants play a key role in checking correctness of code, e.g., a model checker can use an invariant, e.g., acyclicity of a binary tree, that is written in the form of an assertion to search for program executions that violate it, e.g., erroneously introduce a cycle in the structure. Traditionally, the properties are written manually by the users. However, writing them manually can itself be error-prone, which can lead to false alarms or missed bugs. This paper presents a controlled experiment on applying a suite of off-the-shelf machine learning (ML) tools to learn properties of dynamically allocated data structures that reside on the program heap. Specifically, we use 10 data structure subjects, and systematically create training and test data for 6 ML methods, which include decision trees, support vector machines, and neural networks, for binary classification, e.g., to classify input structures as valid binary search trees. The study reveals two key findings. One, most of the ML methods studied – with off-the-shelf parameter settings and without fine tuning – achieve at least 90% accuracy on all of the subjects. Two, high accuracy is achieved even when the size of the training data is significantly smaller than the size of the test data. We believe future work can utilize the learnt invariants to automate dynamic and static analyses, thereby enabling advances in machine learning to further enhance software testing and verification techniques.

Keywords: Data structure invariants · Machine learning · Korat

1 Introduction

Data structure invariants are properties that the data structures in a program must satisfy in valid states, e.g., a binary search tree implementation must create structures that are trees, i.e., contain no cycles, and consist of keys that appear in the tree in the correct search order. In object-oriented programs such invariants are termed class invariants and are expected to hold in all publicly-visible states [28,34].

Data structure invariants play a key role in testing and verification. For example, when written as assertions they enable a number of assertion-based checking

F. Biondi et al. (Eds.): SPIN 2019, LNCS 11636, pp. 226–243, 2019.
https://doi.org/10.1007/978-3-030-30923-7_13

techniques. To illustrate, in software testing, they serve as test assertions as well as a basis of automated test generation [4,27]; in model checking, they serve as target assertions that a model checker can try to violate, i.e., find a program execution that leads to an assertion violation [20,23,33,47]; in runtime verification, they provide a basis for error recovery using data structure repair [11,13]; and in static analysis, they enable deep semantic checking [8,24,35,40,42,48].

Data structure invariants are often written manually by users who want to utilize them for automated testing or verification. However, writing complex invariants manually itself can be error-prone and errors in invariants can lead to false alarms or undetected faults. To reduce the burden on the user to write invariants, researchers have developed several techniques for automatically creating invariants using various forms of analyses. While a vast majority of the techniques utilize static or dynamic analysis [10,12,14,26,29,32,35,40,42,43,48], a few techniques have leveraged machine learning methods to characterize invariants [16,30] and serve as a basis for our work.

This paper presents a controlled experiment on applying a suite of off-the-shelf machine learning (ML) tools to learn invariants of dynamically allocated data structures. Specifically, we use 6 ML methods that include four methods based on decision trees [39], as well as support vector machines [9] and multilayer perceptrons [36]. As data structure subjects we use structural invariants of 10 data structures that have been studied before in several contexts [4,13,16], including most recently for training binary classifiers using feed-forward artificial neural networks [16].

The subjects were introduced in the public distribution of the automated test input generator Korat [1,4] and were originally developed for the purpose of evaluating Korat's input generation. Each data structure contains a Java method called *repOk* that implements an executable check for the properties that represent the corresponding structural invariants (and a variety of other methods). Given a *repOk* method and a bound on the input size, e.g., 5 nodes for a binary search tree, Korat performs a backtracking search over the space of all candidate inputs (up to the size bound) for *repOk* to systematically enumerate all inputs for which *repOk* returns true. For increased efficiency, Korat only considers non-isomorphic candidates. During its search, Korat typically inspects each candidate by running *repOk* on it to get feedback for pruning the search, and as a result outputs only the *valid* inputs, i.e., inputs for which *repOk* returns true.

Our study methodology is as follows. For each data structure subject invariant and ML model, we first create training and test data, then we train the ML model using the training data, and finally we evaluate it using the test data. To create the training/test data, we use Korat to exhaustively explore the bounded input space and create every valid input. The set of all valid inputs forms the positive samples and a subset of invalid inputs inspected by Korat forms the negative samples. In general, for complex structural properties, the number of valid structures is much smaller than the number of invalid structures. Therefore, to avoid training an incorrect model that simply learns to predict false with

high probability, we use *balanced* sets of samples such that there are the same number of positive and negative samples. To study how *learnable* the invariants are we vary the ratio of training and test data from 75 to 25 respectively, which is common in the field of machine learning, to 10 to 90 respectively, which allow us to study the setting where the training data is relatively scarce.

The study reveals two key findings. One, most of ML methods studied – with off-the-shelf parameter settings and without fine tuning – achieve at least 90% accuracy on all of the subjects. Two, the accuracy is achieved even when the size of the training data is significantly smaller than the size of the test data. We find the results quite encouraging and believe machine learning methods hold much promise in developing new techniques for more effective software analysis.

The training and test/evaluation datasets used in our study are publicly available at: https://github.com/muhammadusman93/Spin2019KoratML.

2 Background: Korat and Learning

This section provides the necessary background on the Korat test input generator [4] and basic machine learning models that we use in our study.

2.1 Korat

Korat is a framework for automatic test input generation for Java programs. It takes as input a Java predicate, termed *repOk* method, and a finitization on the input domain, and generates all possible inputs for which the predicate returns true. Korat repeatedly executes *repOk* on candidate inputs, monitors the object fields accessed by *repOk* for each input, and uses this information to create next candidates to consider. Korat implements a backtracking search that prunes large parts of the input space while preserving the completeness of the search and correctness of the generated valid test input. Moreover, Korat generates only non-isomorphic inputs and does not consider any isomorphic candidates during search, which significantly reduces the number of generated inputs and time overhead.

To illustrate, Fig. 1 shows the `BinaryTree` class, including the `repOk` predicate and finitization `finBinaryTree`. The binary tree has a root field of type `Node` and a `size` field that is a primitive integer. The `Node` class declares a `left` field and a `right` field, representing the left child and the right child of the node. The method `repOk` checks if its input does not have any cycle and has the correct value for size. `repOk` returns `true` if the checked property holds and `false` otherwise. The finitization method `finBinaryTree` specifies a bound on the total number of nodes, and the min and max values for size.

The Korat search internally represents each candidate input structure using a candidate vector of integer indices whose length depends on the finitization and elements that represent object fields. Each element of the candidate vector indexes into an appropriate domain of values for the corresponding field. To illustrate, for a finitization of up to 3 nodes (Node 1, Node 2, and Node 3) and

```
class BinaryTree {
  static class Node {
    Node left, right; }

  Node root;
  int size;

  boolean repOk() {
    if (root == null) return size == 0;
    // checks that tree has no cycle
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
      Node current = (Node) workList.removeFirst();
      if (current.left != null) {
        if (!visited.add(current.left)) return false;
        workList.add(current.left);
      }
      if (current.right != null) {
        if (!visited.add(current.right)) return false;
        workList.add(current.right);
      }
    }
    // checks that size is consistent
    return (visited.size() == size); }

  static IFinitization finBinaryTree(int size) {
    return finBinaryTree(size, size, size); }

  static IFinitization finBinaryTree(int nodesNum, int minSize,
                                     int maxSize) {
    IFinitization f = FinitizationFactory.create(BinaryTree.class);
    IObjSet nodes = f.createObjSet(Node.class, nodesNum, true);
    f.set("root", nodes);
    f.set("size", f.createIntSet(minSize, maxSize));
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    return f; }}
```

Fig. 1. BinaryTree repOk and finitization

size equal to 3, Korat creates a candidate vector of length 8: index 0 represents the value of the root field; index 1 represents the size (and its value is fixed as 0 since size is allowed to take only one value, i.e., 3); indexes 2 and 3 represent the left and right children of Node 1 respectively; likewise indexes 4, 5 and 6, 7 represent the left/right children of Node 2 and Node 3 respectively. The

value of each index that represents a node ranges from 0 to 3, representing 4 possibilities: [`null`, Node 1, Node 2 and Node 3]. This finitization defines a bounded exploration space of size $4 \times 1 \times (4 \times 4)^3 = 16,384$ since the tree root and each of left and right fields of each of the 3 nodes have 4 possible values, and the tree size is fixed to 1 value.

The Korat search generates the following candidate vectors for a binary tree using this finitization:

```
0 0 0 0 0 0 0 0 :: 0 1
1 0 0 0 0 0 0 0 :: 0 2 3 1
1 0 0 1 0 0 0 0 :: 0 2 3
1 0 0 2 0 0 0 0 :: 0 2 3 4 5 1
1 0 0 2 0 1 0 0 :: 0 2 3 4 5
1 0 0 2 0 2 0 0 :: 0 2 3 4 5
1 0 0 2 0 3 0 0 :: 0 2 3 4 5 6 7 1 ***
1 0 0 2 0 3 0 1 :: 0 2 3 4 5 6 7
.....................................
1 0 2 3 1 0 0 0 :: 0 2 3 4
1 0 2 3 2 0 0 0 :: 0 2 3 4
1 0 2 3 3 0 0 0 :: 0 2 3 4
```

Each row shows two entities separated by `::`. The first entity is the candidate vector and is shown before `::`. The second entity is field access ordering and is shown after `::`. Valid structures are marked by `***`.

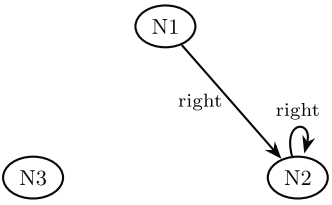

Fig. 2. Invalid binary tree

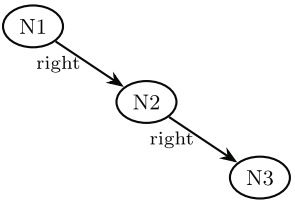

Fig. 3. Valid binary tree

To illustrate, the candidate vector [1 0 0 2 0 2 0 0] represents an invalid binary tree as shown in Fig. 2. The first index states that Node 1 is the root node. The left child of Node 1 is `null` and the right child of Node 1 is Node 2. Similarly, the left child of Node 2 is `null` and the right child of Node 2 is Node 2 itself. Both children of Node 3 are `null`. Thus, the candidate vector represents an invalid binary tree because Node 2 has a self-loop (cycle).

Another example candidate vector [1 0 0 2 0 3 0 0] represents a valid binary tree as shown in Fig. 3. This candidate vector shows that Node 1 is the root node. The left child of Node 1 is `null` and the right child of Node 1 is Node 2. Similarly, the left child of Node 2 is `null` and the right child of Node 2 is Node

3. Both children of node 3 are `null`. Since the binary tree has no cycle, and it has size 3 with 3 nodes reachable from the root, the binary tree is valid.

For this finitization, Korat creates and inspects 63 candidate structures (out of 16384 total candidates while pruning the rest), and outputs 5 of them as valid binary trees with 3 nodes. Korat search breaks isomorphisms, which helps to reduce the number of structures to be explored and generated, thus speeding up the search – note, none of the structures explored by Korat are isomorphic. To illustrate Korat's backtracking search, when Korat finds that the candidate vector [1 0 0 2 0 2 0 0] makes the `repOk` returns false and the last accessed field is the right child of Node 2, it simply increases the value of index 5 (from 2 to 3) and point the right child of Node 2 to Node 3. Korat knows that the left and right children of Node 3 do not affect the result of `repOk` since those fields are not read by `repOk` for the given candidate and thus can be ignored for this combination of values for fields accessed. This pruning helps Korat remove a lot of invalid structures in practice.

2.2 Machine Learning Models

The machine learning models used in the study are Decision Tree (DT) Classifier [39], ensemble Decision Tree Classifiers (including Random Forest Tree Classifier (RFT) [22], Gradient Boosting Tree Classifier (GBDT) [18] and Adaboost Decision Tree Classifier (ADT) [17]), Support Vector Machine (SVM) [9], and Multi-Layer Perceptron (MLP) [36]. We used Python programming language and Scikit-Learn library [2] to implement these machine learning models.

2.2.1 Decision Tree Classifiers

DT classifier takes a tree as a classifier where each leaf node represents the label of the class, and each intermediate node represents a test on a feature. DT is easy to train and can handle qualitative features without using dummy encoding. However, DT is not good in understanding complex relationships between features and is sensitive to the changes in training data.

2.2.2 Ensemble Decision Tree Classifiers

RFT classifier is based on the bootstrap aggregating (Bagging) technique. The underlying idea is to create multiple decision trees and then combine their results to predict the final classification labels. This technique reduces variance of the model and also does not increase bias, and usually overcomes the problem of over-fitting if sufficient number of decision trees are used. GBDT classifier uses a differentiable loss function and creates a strong model using many weak models. ADT classifier makes use of the results of previous trees to select the next trees so that the focus can be shifted on samples which are much harder to classify. Here, multiple weak learners work together to make a strong classifier. After every iteration, weights are assigned to the training samples and higher weight samples get more priority in later trees.

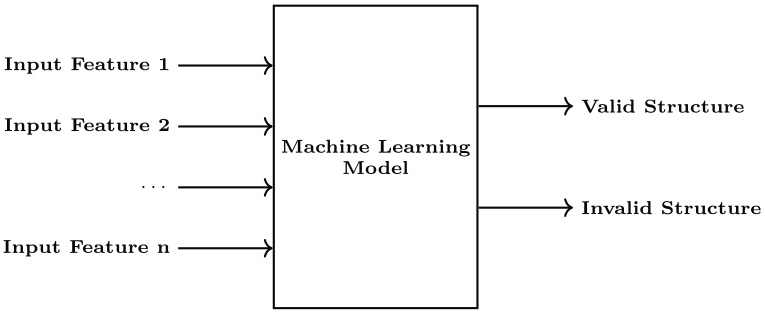

Fig. 4. Architecture of the experimental setup

2.2.3 Support Vector Machine

SVM is a non-probabilistic binary linear classifier and assigns each training sample to one of the two categories. They use a technique called kernel trick in which the data is mapped to a higher dimension making it linearly separable. This makes SVM useful in high dimensional spaces, and flexible with different kernel functions. However, when the number of features is more than the training samples, it is critical to choose the right kernel function and regularization parameters.

2.2.4 Multi-layer Perceptron

MLP is a type of artificial neural network consisting of multiple layers. The first layer is called the input layer and the last layer is called the output layer, with at least one hidden layer in between. The algorithm applies back propagation technique for training, using different non-linear activation functions like tanh and relu. MLP are fully connected and each connection has a weight which is updated during the training phase usually by Stochastic Gradient Descent [41] approach. The main advantage of MLP is its excellent performance in classification, although more training data is needed which makes the training phase time-consuming.

2.3 Encoding Data Structures as Inputs to ML Models

The Korat candidate vector representation provides an immediate encoding for input structures as inputs for binary classification using machine learning models as shown in recent work [16]. Once the finitization is defined, the length of the candidate vector and the ranges of values each element in the vector are precisely defined. Thus, if the candidate vector has length n, the machine learning model for binary classification has n input features and one output (in $\{0, 1\}$), which represents whether the input structure is valid (1) or not (0).

Figure 4 illustrates the experimental setup.

3 Study Subjects

As data structure subjects we select 10 subjects from the standard Korat distribution [1,4]. The subjects include a variety of textbook data structures implemented in Java: singly-linked lists (SLL), sorted lists (SL), binary trees (BT), binary search trees (BST), red-black trees (RBT), binary heaps (BH), heap arrays (HA), Fibonacci heaps (FH), disjoint sets (DS), and directed acyclic graphs (DAG).

4 Study Methodology

In this section, we present our study methodology including generation of training and test data using Korat, selection of finitization bounds for Korat, selection of positive and negative samples, and learning with machine learning classifiers.

4.1 Generation of Training and Test Data

For each data structure invariant, we use Korat to generate the training and test data for the machine learning models. Inputs that satisfy the invariants are termed *positive data* and inputs that violate the property are termed *negative data*. The inputs generated by Korat serve as positive data and the candidates explored by Korat but found to violate an invariant serve as a pool for selecting negative data. Given the structural complexity of all our subjects, the number of valid structures is much smaller than the number of invalid structures. For each subject, we create balanced [38] pools of positive and negative data. Section 4.2 explains how we select the finitization bounds in view of the learning quality of the ML models. Section 4.3 further describes how we select positive and negative data.

Each data sample consists of a candidate vector whose elements serve as features, and a binary label that specifies whether the candidate is valid or invalid. Since different data structures have different fields and may have different finitizations, the positive and negative data for different subjects may vary in length. However, for one subject, each data sample has the same length. To illustrate, for the binary tree subject, for a finitization that allows 10 nodes, the candidate vector has length 22 where the first two fields are root and size of the tree and each of the subsequent two fields represent left and right child of one of the 10 nodes. Thus, each data sample has 23 entries, 22 that are features defined by the Korat candidate vector and 1 that defines whether the candidate is valid or invalid.

4.2 Selection of Finitization Bounds

The finitization bound chosen for each structure determines the space of input candidates that Korat searches and the number of valid structures it creates. Note that different data structures can have very different numbers of valid

structures for the same size, e.g., the number of binary search trees with n nodes is much greater than the number of red-black trees for n nodes due to the height-balance property of red-black trees.

Our main criteria for setting the finitization bound for Korat was to select the smallest bound such that there were sufficient amount of training and test data for the application of machine learning models and at the same time if manual tuning of parameters is needed, the amount of data does not create an impractical problem. Specifically, we chose the bound of at least 10,000 positive data, i.e., valid structures, for all but one of our subjects.

As we explain in Sect. 4.3, we select the same number of negative samples as positive samples, so we have at least 20,000 samples for each data structure invariant (except one). To illustrate, we have to set the finitization bound of Binary-Tree property to 10 nodes, which generates 16796 positive samples.

For one of our subjects, namely red-black trees, we chose a finitization bound of 9 nodes, which gave fewer than 10,000 valid solutions since generation for a higher bound timed out. Specifically, we used the bound of 9,0,9,9, which specifies the number of nodes, the minimum size of the tree, the maximum size of tree and the number of unique integer keys in the tree respectively. For this bound, there are 6753 positive samples and 2262280 negative samples. The positive samples consist of all non-isomorphic red-black trees that can be formed with up to 9 nodes where each node contains a key from a set of 9 unique integer values. Table 1 shows for each subject, the finitization bound (as provided to the finitization method of the subject using `--args` command line option), the size of the state space for the given finitization, the number of valid structures found by Korat, the number of invalid structures explored by Korat, and finally the total number of structures explored by Korat.

Table 1. Candidate structures explored by Korat for each data structure subject.

Subject	Finitization bound	State space	Valid explored	Invalid explored	Total explored
SLL	0,9,10,10	2^{75}	26443	500868	527311
BST	8,0,8,0,7	2^{81}	12235	3613742	3625977
BH	7	2^{109}	107416	154372	261788
BT	10	2^{72}	16796	798304	815100
SL	0,8,9,9	2^{96}	24310	150962	175272
HA	6	2^{23}	13139	51394	64533
DS	5	2^{39}	41546	372309	413855
RBT	9,0,9,9	2^{135}	6753	2262280	2269033
FH	5	2^{82}	52281	112084	164365
DAG	6	2^{108}	19696	185197	204893

4.3 Selection of Positive and Negative Samples

The positive samples consist of every (non-isomorphic) valid structure generated by Korat for the chosen finitization bound. To balance the dataset, we randomly select the same amount of negative samples as the positive ones from the full negative dataset that consists of each candidate Korat explored but found to be invalid. To illustrate, the Disjoint-Set invariant had 41546 positive samples and 372309 negative samples. We kept all of the 41546 positive samples and randomly selected 41546 samples from 372309 negative samples.

4.4 Learning with Machine Learning Classifiers

A key factor in applications of machine learning models is the ratio of training and test data. Traditionally, ratios of 80:20 or 75:25 for training:test are commonly used. We use 4 different ratios in our study. Specifically, we performed experiments using each of the following training:test ratios – 75:25, 50:50, 25:75, and 10:90. Thus, on one extreme, we explore the more traditional setting where 75% of data are used for training and 25% are used for evaluation, and on the other extreme, we explore the unconventional setting of using just 10% data for training and 90% for evaluation. As is common practice in evaluating ML models, our training and test data had no overlap. Moreover, due to the use of Korat, not only is there no intersection in the training and test datasets but also the two datasets don't contain isomorphic structures.

We ran experiments using base ML models taken off-the-shelf, and also using manually tuned models. The tuned models performed only slightly better than base models but the overhead in finding tuned hyper-parameters outweighed the increase in accuracy. Therefore, we report the results of base models only in Tables 2 and 3.

We report counts of True Negatives (TN), False Positives (FP), False Negatives (FN) and True Positives (TP) in Tables 2 and 3. True Negative is when the ground truth label is 0 and the classifier correctly predicted label 0. False Positive is when the ground truth label is 0 but the classifier wrongly predicted label 1. False Negative is when the ground truth label is 1 but the classifier wrongly predicted label 0. True Positive is when the ground truth label is 1 and the classifier correctly predicted label 1. In addition, we use four metrics to report the results of the classification: Precision, Recall, Accuracy and F1 score. Precision is calculated as $\frac{TP}{TP+FP}$. Recall is calculated as $\frac{TP}{TP+FN}$. Accuracy is calculated as $\frac{TP+TN}{TP+TN+FP+FN}$. F1 score is calculated as $\frac{2*Precision*Recall}{Precision+Recall}$.

5 Experimental Results

Experiments were performed with training data percentage of 10%, 25%, 50%, and 75%, and in each case the rest of the data was used for testing, i.e., evaluation of accuracy. In this section, we included detailed results obtained using 10% training data (Tables 2 and 3) and the remaining detailed results are included in

Table 2. Classification results for 10:90 training:test ratio

Property	Model	TN	FP	FN	TP	Accuracy	Precision	Recall	F1
SLL	DT	23728	37	39	23794	0.9984	0.9984	0.9984	0.9984
	RFT	23659	106	25	23808	0.9972	0.9956	**0.9990**	0.9973
	GBDT	23729	36	24	23809	**0.9987**	**0.9985**	**0.9990**	**0.9987**
	ABT	22402	1363	396	23437	0.9630	0.9450	0.9834	0.9638
	SVM	23196	569	24	23809	0.9875	0.9767	**0.9990**	0.9877
	MLP	23691	74	24	23809	0.9979	0.9969	**0.9990**	0.9979
BST	DT	10092	921	865	10145	0.9189	0.9168	0.9214	0.9191
	RFT	10258	755	580	10430	0.9394	0.9325	0.9473	0.9399
	GBDT	10149	864	192	10818	**0.9521**	0.9260	**0.9826**	**0.9535**
	ABT	10030	983	324	10686	0.9407	0.9158	0.9706	0.9424
	SVM	10325	688	1630	9380	0.8947	0.9317	0.8520	0.8900
	MLP	10337	676	380	10630	**0.9521**	**0.9402**	0.9655	0.9527
BH	DT	96447	211	155	96536	0.9981	0.9978	0.9984	0.9981
	RFT	96258	400	215	96476	0.9968	0.9959	0.9978	0.9968
	GBDT	95902	756	382	96309	0.9941	0.9922	0.9960	0.9941
	ABT	93536	3122	1958	94733	0.9737	0.9681	0.9797	0.9739
	SVM	96391	267	50	96641	0.9984	0.9972	**0.9995**	0.9984
	MLP	96523	135	123	96568	**0.9987**	**0.9986**	0.9987	**0.9987**
BT	DT	14979	180	51	15023	**0.9924**	**0.9882**	0.9966	**0.9924**
	RFT	14520	639	607	14467	0.9588	0.9577	0.9597	0.9587
	GBDT	14774	385	194	14880	0.9808	0.9748	0.9871	0.9809
	ABT	13369	1790	0	15074	0.9408	0.8939	**1.0000**	0.9440
	SVM	10467	4692	4996	10078	0.6796	0.6823	0.6686	0.6754
	MLP	14323	836	499	14575	0.9558	0.9458	0.9669	0.9562
SL	DT	21687	154	83	21834	**0.9946**	**0.9930**	0.9962	**0.9946**
	RFT	21306	535	216	21701	0.9828	0.9759	0.9901	0.9830
	GBDT	21262	579	160	21757	0.9831	0.9741	0.9927	0.9833
	ABT	18212	3629	3476	18441	0.8376	0.8356	0.8414	0.8385
	SVM	21345	496	12	21905	0.9884	0.9779	**0.9995**	0.9885
	MLP	21537	304	15	21902	0.9927	0.9863	0.9993	0.9928

the GitHub repository and summarized here due to space limitation. We choose to include 10% here because it is the most interesting case as we train on a relatively small percentage of data and still are able to classify the data structure invariants with surprisingly high accuracy. The key results are as follows.

For 10% training data ratio (i.e., 90% test data), the maximum accuracy for the subject invariants was 99.87%, which was achieved for the binomial heap invariant using multi-layer perceptrons (MLPs). The minimum accuracy was

Table 3. Classification results for 10:90 training:test ratio

Property	Model	TN	FP	FN	TP	Accuracy	Precision	Recall	F1
HA	DT	11735	99	46	11771	**0.9939**	**0.9917**	**0.9961**	**0.9939**
	RFT	11523	311	351	11466	0.9720	0.9736	0.9703	0.9719
	GBDT	11218	616	95	11722	0.9699	0.9501	0.9920	0.9706
	ABT	8852	2982	2812	9005	0.7550	0.7512	0.7620	0.7566
	SVM	9993	1841	536	11281	0.8995	0.8597	0.9546	0.9047
	MLP	10439	1395	576	11241	0.9167	0.8896	0.9513	0.9194
DS	DT	33595	3700	3053	34435	0.9097	0.9030	0.9186	0.9107
	RFT	33730	3565	2820	34668	**0.9146**	**0.9068**	0.9248	**0.9157**
	GBDT	31978	5317	2623	34865	0.8938	0.8677	0.9300	0.8978
	ABT	30079	7216	8561	28927	0.7890	0.8003	0.7716	0.7857
	SVM	30595	6700	2352	35136	0.8790	0.8399	**0.9373**	0.8859
	MLP	32849	4446	2350	35138	0.9091	0.8877	**0.9373**	0.9118
RBT	DT	5807	276	107	5966	0.9685	0.9558	0.9824	0.9689
	RFT	5865	218	65	6008	0.9767	**0.9650**	0.9893	0.9770
	GBDT	5826	257	17	6056	**0.9775**	0.9593	**0.9972**	**0.9779**
	ABT	5836	247	32	6041	0.9770	0.9607	0.9947	0.9774
	SVM	5849	234	155	5918	0.9680	0.9620	0.9745	0.9682
	MLP	5848	235	84	5989	0.9738	0.9622	0.9862	0.9741
FH	DT	45893	1063	1217	45933	0.9758	**0.9774**	0.9742	0.9758
	RFT	44078	2878	3489	43661	0.9323	0.9382	0.9260	0.9320
	GBDT	42326	4630	2913	44237	0.9198	0.9053	0.9382	0.9214
	ABT	37152	9804	9024	38126	0.7999	0.7955	0.8086	0.8020
	SVM	40973	5983	2187	44963	0.9132	0.8826	0.9536	0.9167
	MLP	45885	1071	1100	46050	**0.9769**	0.9773	**0.9767**	**0.9770**
DAG	DT	16162	1579	966	16746	**0.9282**	**0.9138**	**0.9455**	**0.9294**
	RFT	15708	2033	1852	15860	0.8904	0.8864	0.8954	0.8909
	GBDT	15000	2741	1638	16074	0.8765	0.8543	0.9075	0.8801
	ABT	14560	3181	3266	14446	0.8182	0.8195	0.8156	0.8176
	SVM	14296	3445	2013	15699	0.8460	0.8200	0.8863	0.8519
	MLP	15677	2064	2010	15702	0.8851	0.8838	0.8865	0.8852

75.50% for the heap array invariant using Adaboost trees. Overall, decision trees (DTs) performed the best on data structure invariants whereas Adaboost trees (ABTs) performed the worst for the invariants studied. DT average accuracy is 96.79%; random forest (RFT) average accuracy is 95.61%; gradient boosting tree (GBDT) average accuracy is 95.46%; ABT average accuracy is 87.95%; support vector machine (SVM) average accuracy for Korat is 90.54%; and MLP average accuracy is 95.59%.

For 25% training data ratio (i.e., 75% test data), the results observed were as follows. The maximum accuracy for the subject invariant was 99.99%, which was achieved for the Singly-linked list invariant using MLP. The minimum accuracy was 71.60% for the sorted list invariant using SVM. Overall, decision trees performed the best on data structure invariants whereas Adaboost Trees performed the worst of the models studied. DT average accuracy is 98.33%; RFT average accuracy is 97.34%; GBDT average accuracy is 95.55%; ABT average accuracy is 88.14%; SVM average accuracy is 92.30%; and MLP average accuracy is 97.87%.

For 50% training data ratio (i.e., 50% test data), the results observed were as follows. The maximum accuracy for the subject invariant was 99.98%, which was achieved for the heap array invariant using DT and binomial heap invariant using MLP. The minimum accuracy was 75.13% for the binary tree invariant using SVM. Overall, decision trees performed the best on data structure invariants whereas Adaboost trees performed the worst of the models studied. DT average accuracy is 98.96%; RFT average accuracy is 98.16%; GBDT average accuracy is 95.78%; ABT average accuracy is 88.16%; SVM average accuracy is 93.64%; and MLP average accuracy is 98.92%.

For 75% training data ratio (i.e., 25% test data), results observed as follows. The maximum accuracy for the subject invariant was 100%, which was achieved for the heap array invariant using DT and sorted list using MLP. The minimum accuracy was 78.08% for the binary tree invariant using Adaboost Tree. Overall, decision trees performed the best on data structure invariants whereas Adaboost Trees performed the worst of the models studied. DT average accuracy is 99.27%; RFT average accuracy is 98.58%; GBDT average accuracy is 95.65 %; ABT average accuracy is 88.28%; SVM average accuracy is 94.45%; and MLP average accuracy is 99.24%.

Overall, from the study we conclude that decision trees are quite good in predicting structurally complex properties whereas Adaboost trees have the least accuracy. We also observe that overall the accuracy ranges from a low of 71.60% to a high of 100.00%.

All experiments were performed on an Intel i7-4700MQ (2.40 GHz) processor with 8 GB of RAM.

6 Threats to Validity

In our experiments, we use a fixed size for each subject. The ML classifiers may perform worse for smaller sizes of subjects due to less available training data and better for larger sizes of subjects due to more available training data.

The negative examples generated by Korat makes the irrelevant fields their default values because setting those fields to any value does not change the false result of *repOk*. So those examples are canonical compared to the entire negative example space. As a consequence, our results may not hold for other negative examples.

As explained in Sect. 2.1, the training data from Korat was always correctly labeled.Thus, this data had no noise. However, in practical situations, the train-

ing data does not have this quality. Normally, training data has some samples which are labeled wrong or have missing values. This situation did not occur here and this is one of the main reason behind high accuracy values observed during the course of this study.

Another threat to validity is the undersampling technique used in this study. We can see that the negative cases had a much larger state space and we have to do undersampling to make the classes balanced. Also it is impossible to generate all the negatives in some cases. For example, the structures explored for the Red-Black Tree invariant were 2269033. We tried to randomly sample the negative samples but more work should be done in future to find a better way of dealing with imbalanced classes and dealing with large state space.

7 Related Work

A number of research projects introduced the use of machine learning methods in learning properties of software systems [5,7,16,19,30,44]. In the specific context of structural properties of data structures, to our knowledge, Malik [30] first introduced the use of a machine learning method, namely support vector machines, for characterizing the properties, specifically by utilizing graph spectra [6]. Most recently, Molina et al. [16] introduced the first use of feed-forward artificial neural networks as binary classifiers for data structure properties and showed their trained networks had high accuracy and worked better than an approach [14] for using dynamic analysis for detecting likely program invariants.

Our study is closest to Molina et al.'s work and extends it in three important directions. One, we evaluate 6 machine learning models, including decision trees and support vector machines, that were not studied in their work that only used neural networks. Two, we use 4 data structure subjects that were not in their study as well as 6 subjects that were in their study. Three, we study several different ratios of test/training data whereas their study did not consider any specific test/training ratio, rather the ratio in their study was driven by the training data generated by the test generation tool Randoop [37]. Moreover, we have no overlap between test and training data whereas in their study there was up to >50% overlap for positive cases (e.g., for binary search trees and red-black trees) and for each subject the test data contained all of the training data. Overall, the results of our study generally corroborate their findings, but in addition, enhance them along new dimensions.

There is a rich body of work on using dynamic analysis and static analysis in detecting and generating (likely) program invariants [12,14,26,32,35,40,42,43,48]. Ernst [14,15] is a widely studied tool for generating likely program invariants. The key idea in Daikon is to use a collection of pre-defined property templates and observe program states at control points of interest to check which of the properties consistently hold at those points, and then to consider those as likely invariants. While Daikon is quite effective at properties over integers and arrays, its effectiveness is relatively low for structural properties. Deryaft [29] followed Daikon's spirit to introduce a technique for generating likely structural invariants

and can handle complex data structures. However, a key issue with the Daikon family of techniques is that they require a collection of property templates and can only create invariants based on those properties (and boolean connections among them).

There is a large body of work on program synthesis [3,21,31] and sketching [46] that is applicable to invariant generation in principle. We believe machine learning methods can also be helpful in improving some of these techniques, e.g., by guiding the search in the space of candidate programs [25,45].

8 Conclusion

This paper presented a controlled experiment on applying a suite of off-the-shelf machine learning (ML) tools to learn properties of dynamically allocated data structures that reside on the program heap. Specifically, we used 10 data structure subjects, and systematically created training and test data for 6 ML methods, which include decision trees, support vector machines, and neural networks, for binary classification, e.g., to classify input structures as valid binary search trees. The study had two key findings. One, most of ML methods – with off-the-shelf parameter settings and without fine tuning – achieves at least 90% accuracy on all of the subjects. Two, the accuracy is achieved even when the size of the training data is significantly smaller than the size of the test data. We believe machine learning models offer a promising approach to characterize data structure invariants.

Acknowledgments. This research was partially supported by the US National Science Foundation under Grant Nos. CCF-1704790 and CCF-1718903.

References

1. Korat GitHub repository. https://github.com/korattest/korat
2. Scikit-Learn Library. https://scikit-learn.org/stable/. Accessed 18 Aug 2019
3. Bodik, R.: Program synthesis: opportunities for the next decade. In: 20th ACM SIGPLAN International Conference on Functional Programming, p. 1 (2015)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ACM SIGSOFT 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: 18th IEEE International Symposium on Software Reliability, pp. 137–146 (2007)
6. Brouwer, A.E., Haemers, W.H.: Spectra of Graphs. Springer, New York (2012). https://doi.org/10.1007/978-1-4614-1939-6
7. 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 (FMCAD), pp. 76–83 (2017)
8. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: 40th Design Automation Conference, (DAC), pp. 368–371 (2003)

9. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
10. Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: dynamic symbolic execution for invariant inference. In: 30th International Conference on Software Engineering, pp. 281–290 (2008)
11. Demsky, B., Rinard, M.C.: Automatic detection and repair of errors in data structures. In: Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, pp. 78–95 (2003)
12. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 443–456 (2013)
13. Elkarablieh, B., Garcia, I., Suen, Y.L., Khurshid, S.: Assertion-based repair of complex data structures. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 64–73 (2007)
14. 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)
15. Ernst, M.D., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
16. Molina, F., Degiovanni, R., Ponzio, P., Regis, G., Aguirre, N., Frias, M.F.: Training binary classifiers as data structure invariants. In: International Conference on Software Engineering (ICSE), May 2019
17. 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)
18. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Ann. Stat. **29**(5), 1189–1232 (2001)
19. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 499–512 (2016)
20. Godefroid, P.: Model checking for programming languages using VeriSoft. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 174–186 (1997)
21. Gulwani, S., Dimensions in program synthesis. In: 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, pp. 13–24 (2010)
22. Ho, T.K.: Random decision forests. In: Third International Conference on Document Analysis and Recognition, vol. 1 (1995)
23. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional, Boston (2011)
24. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 14–25 (2000)
25. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 215–224 (2010)
26. Jump, M., McKinley, K.S.: Dynamic shape analysis via degree metrics. In: 8th International Symposium on Memory Management (ISMM), pp. 119–128 (2009)
27. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. **16**(8), 870–879 (1990)
28. Liskov, B., Guttag, J.V.: Program Development in Java - Abstraction, Specification, and Object-Oriented Design. Addison-Wesley, Boston (2001)

29. Malik, M., Pervaiz, A., Uzuncaova, E., Khurshid, S.: Deryaft: a tool for generating representation invariants of structurally complex data. In: ACM/IEEE 30th International Conference on Software Engineering (2008)
30. Malik, M.Z.: Dynamic shape analysis of program heap using graph spectra: NIER track. In: 33rd International Conference on Software Engineering (ICSE), pp. 952–955 (2011)
31. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
32. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_31
33. Mera, E., Lopez-García, P., Hermenegildo, M.: Integrating software testing and run-time checking in an assertion verification framework. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 281–295. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02846-5_25
34. Meyer, B.: Class invariants: concepts, problems, solutions. CoRR, abs/1608.07637 (2016)
35. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 221–231 (2001)
36. Murtagh, F.: Multilayer perceptrons for classification and regression. Neurocomputing **2**(5), 183–197 (1991)
37. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering, pp. 75–84 (2007)
38. Provost, F.: Machine learning from imbalanced data sets 101. In: Proceedings of the AAAI 2000 Workshop on Imbalanced Data Sets, vol. 68, pp. 1–3. AAAI Press (2000)
39. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
40. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science (2002)
41. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat. **22**(3), 400–407 (1951)
42. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 105–118 (1999)
43. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 318–329 (2004)
44. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 31, pp. 7751–7762 (2018)
45. Singh, S., Zhang, M., Khurshid, S.: Learning guided enumerative synthesis for superoptimization (2019, under submission)

46. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis (2008)
47. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: Fifteenth IEEE International Conference on Automated Software Engineering (ASE), pp. 3–12 (2000)
48. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 349–361 (2008)