On the Ability to Learn Graph Properties Using Off-the-Shelf Machine Learning Models

Submitted To

Sarfraz Khurshid
Electrical and Computer Engineering Department
University of Texas at Austin

Prepared By

Rohan Garg
Emily Ginsburg
Michael Herrington
Tara Kuruvilla
Raghav Prakash

EE464 Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Spring 2020

CONTENTS

TABLES v

FIGURES vi

EXECUTIVE SUMMARY vii

1.0 INTRODUCTION 1

2.0 DESIGN PROBLEM 2

2.1 Background 2

2.2 Project Goals 3

2.3 Deliverables 4

3.0 DESIGN SOLUTION 5

3.1 Data Generation 6

3.1.1 Java RepOk Methods 6

3.1.2 Python Scripts: Parsing and Splitting 8

3.1.3 GitHub Repository with Persistent Data 9

3.2 Machine Learning Models 9

3.2.1 Python Notebooks of Trained ML Models 11

3.2.2 Charts of ML Model Evaluation Metrics 11

3.3 Integrated Testing Tool 13

3.3.1 JUnit Test Files 13

3.3.2 Chart of Runtime and Memory-Use for JUnit Tests 14

3.4 Final Paper 15

4.0 DESIGN IMPLEMENTATION 15

4.1 Modified Invariants 15

4.1.1 Problem 16

4.1.2 Solution 16
4.2 Modified ML Model Types 17
4.2.1 Problem 17
CONTENTS (Continued)
4.2.2 Solution 18
4.3 Modified Dataset Properties 19
4.3.1 Problem 19
4.3.1 Solution 19
5.0 TEST AND EVALUATION .....
20
5.1 Data Generation Module 20
5.1.1 Method 20
5.1.2 Results 22
5.1.3 Analysis 24
5.2 ML Model Module 24
5.2.1 Method 25
5.2.2 Results 25
5.2.3 Analysis 25
5.3 Integrated Testing Tool Module 26
5.3.1 Method 26
5.3.2 Results 27
5.3.3 Analysis 27
6.0 TIME AND COST CONSIDERATIONS 28
7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN 29
8.0 RECOMMENDATIONS 30
8.1 Recommendations for Future Extensions 31
8.1.1 Creating Larger Datasets 31
8.1.2 Making Korat Improvements 31
8.1.3 Modifying ML Models 32

8.1.4 Finding Additional Applications 32
8.2 Alternative Design and Implementation Decisions 33
8.2.1 Streamlining Model Creation 33
8.2.2 Make Model Training Platform Independent (Java) 33

CONTENTS (Continued)

8.2.3 New model evaluation and re-training methods 34
9.0 CONCLUSION 35
REFERENCES 37
APPENDIX A – TABLE OF ML MODEL METRICS RESULTS ....
A-1

TABLES

1. *Table 1. Technical Specifications for Deliverables Table..17*
2. *Table 2. Machine Learning Models..22*
3. *Table 3. Metrics to Assess ML Models (condensed)...24*
4. *Table 4. Template for Chart of Runtime and Memory-Use for JUnit Tests.......................26*
5. *Table 5. Korat Runtime Metrics per Invariant...34*
6. *Table 6. Chart of Runtime and Memory-Use for JUnit Tests..38*

FIGURES

1. *Figure 1. System Design Diagram...18*
2. *Figure 2. Visualizations of Korat candidate vectors..34*
3. *Figure 3. Diagram of isomorphic reassignment of nodes in a graph................................45*

EXECUTIVE SUMMARY

In our project, our team uncovers the validity of using off-the-shelf machine learning (ML) models, commonly used in industry, to validate graph data structures for testing purposes. Our research aims to increase the efficiency of conventional software testing with trained machine learning models. To prove the validity of implementing ML models as data structure checks, we generate datasets of graphs with varying properties, train ML models to recognize invariants in the graphs, and integrate an end-to-end product using the ML models in a JUnit test suite. With our project, we were able to conclude that certain graph properties are easily learnable and that these models can be packaged into useful runtime testing tools. In section one Introduction, we outline the contents of the paper and describe the motivation of the research to explore automation in testing.

Using the description of prior art as a model for our research, we discuss the goals and deliverables for our project in section two Design Problem. Within section two, The Background portion discusses the research paper, *A Study of Learning Data Structure Invariants Using Off-The-Shelf Tools* [1], which outlines the system design that our project is based on. The Goals section discusses the two overarching objectives of our project: to extend the study from this paper and to test the efficiency of the research in a normal software testing workflow. To complete these goals, we use the Project Specifications section to describe the three main deliverables for our project, consisting of a GitHub Repository, an Integrated Testing Tool with the ML models included in a software testing system, and this Final Paper.

In the third section Design Solution, we discuss all of our tangible work throughout the course of the project, culminating with this paper describing the final system design solution. The design is broadly broken down into three modules: Data Generation, Machine Learning, and Integrated Testing Tool. Our Data Generation module describes all the pieces needed to generate data for the five graph invariants or properties: contains self-loops, K-regularity, acyclicity, density (greater than 0.70), and contains exactly one root. The tools used to produce the data incluse

Korat, a test-input generation tool, to create the raw data, Python to parse the data from Korat, and GitHub to store the clean data for the ML models. The Machine Learning Models module then trains, tests, and evaluates the ML models using the data from the Data Generation module with Python notebooks and the Scikit Python library. Finally, the most accurate models are implemented in an end-to-end software testing flow within JUnit, a common software testing suite, in the Integrated Testing Tool module.

In section four Design Implementation, we highlight three key areas where we faced obstacles including the high complexity of our initial proposed invariants, overhead of data generation, and selection of machine learning models. To combat these we reduced the complexity of our invariants, reduced the number of machine learning models we were using, and modified certain dataset properties to boost performance.

In the fifth section Test and Evaluation, we describe how we tested and evaluated our entire design process and workflow. First, we tested Korat through some theoretical computations as well as visualizations for the Korat graph output. Using the visualizations, we were able to notice how Korat does not put repeated edges or self-loops. Additionally, we were able to test and see how long Korat took for graphs of size n, where we were able to variable change n. Second, we tested our Machine Learning Models module. We trusted that the machine learning models themselves had been tested for accuracy and correctness since they were available through credible library sources that millions of users use. What remained for us to test was our use of the models and specifically how well they were learning these graph invariants. To do this we used various model metrics such as accuracy, precision, recall, Log loss, and Area under Curve. In general, our results consisted of high accuracy/precision. This showed us that most of these invariants are easily learned by the machine learning models. Lastly, we had to test our implementation of these ML models as integrated testing files. Essentially, we wrote a function to convert adjacency matrices to candidate vectors and then ran those vectors through our models to get a binary classification of that vector. To get a better understanding of the success of our module, we measured the time and space cost of each model within their Java wrapper methods.

Next, we describe our time and cost considerations in section six. Our budget was $0 since our research was entirely software based without any need for specialized hardware. In terms of time, we initially planned to work on all invariants concurrently. However, to reduce the number of issues we faced for each invariant, we ran the *oneRoot* invariant through the entire workflow before working on the other invariants. Due to COVID-19, we cut the scope of our project and did not allocate any time towards finding a real-world software system that our tool could easily be integrated into.

In Section seven Safety and ethical Aspects of Design, we describe potential ethical issues with our system design. Since our project is a software testing research effort, there are no immediate physical safety concerns to us as developers or to users. However, the users working with little tolerance for error should know that our ML models may output incorrect answers because the models are trained on biased data and our evaluation proved a 0.1% probability of inaccuracy.

In the penultimate section, we discussed recommendations for groups looking to extend our work as well as alternative design decisions we could have made. We recommend that other groups look into creating larger datasets with the help of supercomputers, improve upon Korat itself, hypertune the ML models for specific applications, and find specific real-world software systems for integration. Some alternative design decisions are making the model creation process more streamlined making our testing tool independent of any specific language (in our case Java). Further, we describe the faults and biases in Korat generation, recommending modifications in both data generation and ML model evaluation.

Lastly, in section nine Conclusion, we summarize the findings from our research project and guide future research in Korat generated ML models for data structure verification.

1.0 INTRODUCTION

In our research, we are uncovering the validity of using off-the-shelf machine learning (ML) models, a common public ML model used in industry, to validate data structures for testing purposes. "Data structures invariants", or properties, "play a key role in testing and verification" [1]. Further, the automation of testing techniques is becoming increasingly more important. Since software is becoming more pervasive in everyday life, industry will produce more code and data. However, the public tolerance for software errors is not increasing proportionally to the amount of code generated. Instead, due to society's increased dependence on technology, the tolerance for error is decreasing. To decrease software errors in an environment with exponentially growing codebases, our team is exploring the application of ML to automate data structure invariant validation. We investigate, specifically, the process of generating graph structures for testing and training models using a testing generator, Korat, as done in the paper, *A Study of Learning Data Structure Invariants Using Off-The-Shelf Tools* [1]. We refer the reader to *Training binary classifiers as data structure invariants* [2] by Molina et. al. for more background on the problem and its motivations.

In the Design Problem section of this report, we provide further background of this original study [1] that provides the foundational processes to generate data as well as train, test, and evaluate ML models. We use the processes described in the paper [1] to evaluate the efficacy of using ML models to validate graph invariants. Then, we test the integration of the models in a common software testing suite, JUnit. In order to complete our two main goals, evaluating ML models that validate graph invariants and integrating the models in a software testing process, our team developed a system design with four modules. The technical deliverables and designs for the four modules -- Data Generation, ML Models, Testing Integration, and Final Paper -- are described in the Design Solution section of the Paper.

The implementation and evaluation of our system design is described in the Design Implementation and Test and Evaluation sections of this paper. In the Design Implementation section, we explain the problems we faced that led us to adjust our system design in the

following three ways: changing the invariants to simpler graph invariants, choosing a subset of ML models, and adjusting the properties of our training data sets. In the Test and Evaluation section, we describe the system's success in creating accurate models for graph invariants.

In the rest of the paper -- Time and Cost Considerations, Safety and Ethical Aspects of the Design, and Recommendations -- we further evaluate our system design as well as our team's efficacy in implementing the design in a timely, cost-effective, and ethical manner. In the Time and Cost Consideration section, we detail how we changed our project schedule to allow for more time to test and iterate. In the Safety and Ethical Aspects section, we describe potential ethical issues in implementing these ML models in industry. In the Recommendation section, we reveal faults in the overall design that may have led to overconfidence of the model's accuracy. Further, we suggest potential solutions and implementation changes on the system to create a more robust and accurate system in further research.

2.0 DESIGN PROBLEM

In this section, we cover the background of our project broken into three subsections: Background, Goals, and Deliverables. The Background section summarizes the research paper, *A Study of Learning Data Structure Invariants Using Off-The-Shelf Tools* [1], which serves as the foundation of our project. The Project Goals section defines two goals-- to extend Dr. Khurshid's study and to test the efficiency of the research in a normal software testing workflow -- that needed to be met for the project to be considered successfully completed. Finally, the Deliverables section outlines and describes three Deliverables -- a GitHub Repository, an integration of the models in a software testing system, and a Final Paper -- that our team created this semester to meet the project goals. For each Deliverable, we detail the technical specifications, which will be further explained in the Design Solution section of the report.

2.1 Background

Our research extends Dr. Khurshid's work on verifying data structures with machine learning

models produced in *A Study of Learning Data Structure Invariants Using Off-The-Shelf Tools* [1]. In the original study, off-the-shelf machine learning models were trained and tested to classify types of data structures with certain structural invariants. An invariant of a data structure is a property about the structure's underlying design and functionality which remains the same regardless of how data are added, removed, or modified. An ML model is both a data structure and a corresponding algorithm which, together, allow computers to "learn" how to quickly and correctly make desired predictions given large sets of data. A model is "off-the-shelf" if it has already been fully implemented and published for public use, usually in a software library. In the study [1], the researchers use Korat, a software tool that generates a vectorized form of all valid and invalid data structures for a given invariant and size specifications, to create a dataset for a small subset of their intended invariants. Then, the researchers use the dataset to develop and tune a library of off-the-shelf machine learning models for each invariant. As a result, the study [1] found that off-the-shelf ML models can quickly learn properties of data structures with high accuracy.

2.2 Project Goals

For our senior design project, our team expanded this work [1] by creating ML models that validate graph invariants. Specifically, we focused on properties of data structures that represent directed graphs. Using the ML models we created, our team additionally explored the applications of the models in standard testing practices. The completion of our project is defined by the fulfillment of the following goals: the project should determine the learnability of graph invariants with ML models and employ the ML models to a software testing strategy.

To fulfill the first goal, determining the learnability of graph invariants, our team mimicked the methodology in the original study [1] to create data in Korat as well as train and test ML models for five new graph invariants. At the end of the process, we graded the efficiency of each ML Model and compiled a list of the best ML Model for each invariant. Our team generated a list of highly accurate ML Models based on the processes described in *A Study of*

Learning Data Structure Invariants Using Off-The-Shelf Tools [1]. However, after further digging, we also discovered weaknesses in the models resulting from implicit biases within the data generation used in the original study. Still, we consider the first goal successfully completed because our evaluation of the advantages and shortcomings for the models in this paper will help direct further research in learning data structures using machine learning.

For the second goal, our team implemented the ML Models in a typical testing suite, JUnit, to explore the benefits of using the models for invariant validation in practice. Our team employed the ML models to verify data structures in a program as part of a runtime-check or unit tests for a simple graph generator Java project. We explored the tools needed to export the off-the-shelf models into JUnit, a widely used testing suite for Java applications in industry, and measured the time and memory efficiency of the tests with ML models. We consider the second goal successfully accomplished because we identified benefits and drawbacks of implementing ML Models for data structure verification in JUnit.

2.3 Deliverables

In order to meet the above goals, our team completed three deliverables -- a GitHub Repository, an integration of the ML Models in a JUnit test, and the Final Paper. The GitHub Repository deliverable holds the code to create the graph invariants, the vectorized data, and the ML model code. By storing the data generation code, the data, and the models, our research on graph invariants can be easily recreated and built upon for further research by Dr. Khurshid and his team. The integration illustrates a small use case on how the ML models trained to validate invariants can be applied to a common software testing system. In this paper, the last deliverable, we document all the methodologies used to create the data, train and test the models, and create the JUnit testing integration. This paper is the most important deliverable because here we evaluate the strengths and weaknesses of our system design to create and integrate ML Models for invariant verification. Further, we suggest improvements and identify points for further research. In Figure 1 below, we outline the technical specifications for the deliverables that will each be comprehensively explained in the next

section of this report.

Table 1. Technical Specifications for Deliverables Table

Deliverable	Technical Specifications
GitHub Repository	• Java Files with RepOk methods For Data Generation • Zip Files of Binary Data Used • Python Files with ML Models • Excel Spreadsheet with Metrics Evaluating Models • All Files described in the JUnit Testing Integration
JUnit Testing Integration	• .pmml Files of ML Models • JUnit Test File with Methods Using ML Models for Verification • Generic Java Graph Class for Testing
Final Paper	• Documentation of Data Generation Method • Documentation of ML Model Generation • Evaluation of ML Models • Evaluation of ML Models Integrated in JUnit Testing

3.0 DESIGN SOLUTION

Our System Design is divided into four modules: Data Generation, Machine Learning Models, Integrated Testing Tool, and this Final Paper. These modules are graphically outlined in Figure 1. This section describes the manner in which the modules are related by using the deliverables of preceding modules as inputs. Additionally, we cover the process and design decisions for each deliverable within the modules.

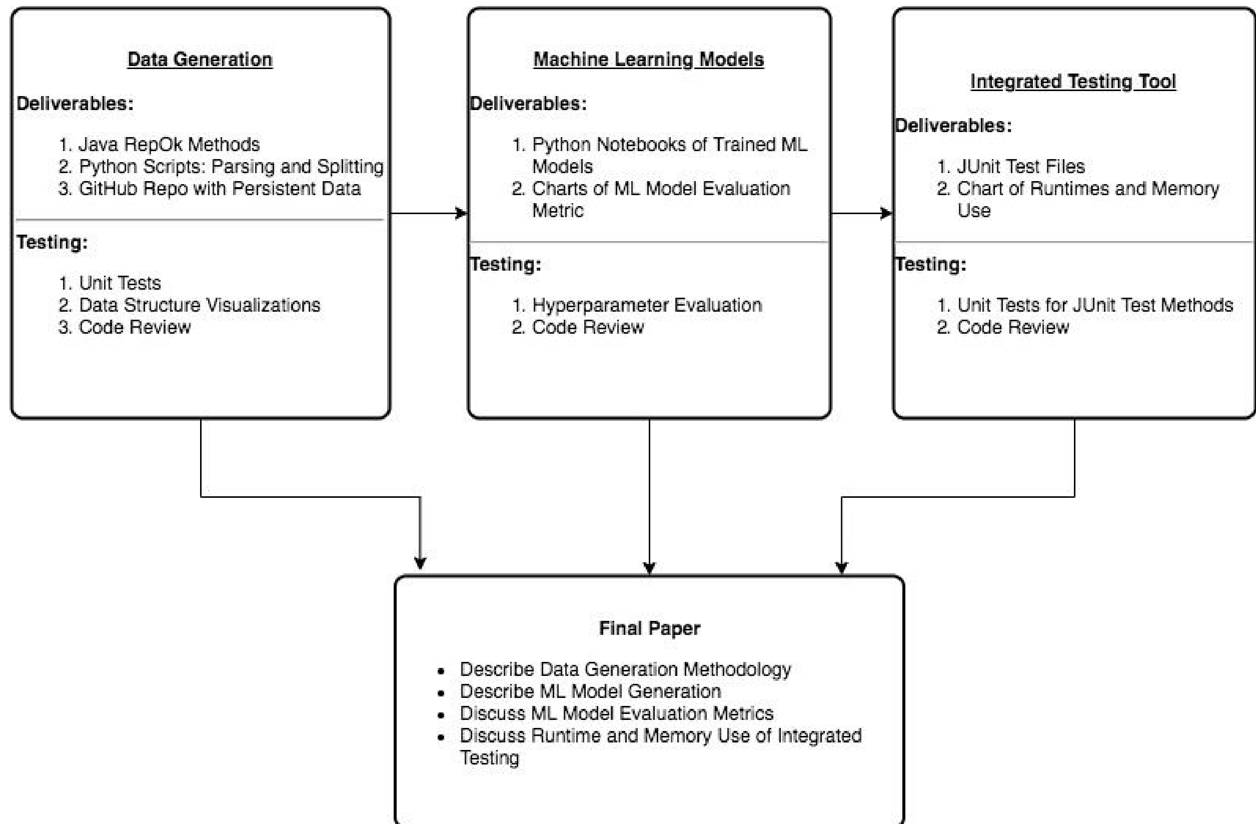

Figure 1. System Design Diagram

3.1 Data Generation

The purpose of the Data Generation module is to create sets of vectorized data structures to be used for training and testing the ML models. This module has three deliverables: the Java RepOk methods for structuring the generated data, the Python scripts for parsing and splitting the data, and a GitHub Repository for organizing and storing all of the data.

3.1.1 Java RepOk Methods

The first deliverable in the Data Generation module is a set of Java RepOk Methods. These methods allowed us to generate a vectorized dataset which represents valid and invalid possibilities of any given data structure. These methods are to be used as inputs to Korat, the software tool we used to automatically generate all, or often a pruned set of, valid and invalid instances of data structures. Korat achieves this by searching through the RepOk method's code for lines and variables which evaluate to any one of multiple possibilities (usually true or

false, or a finite set of numbers), and then producing data structures for each possibility. The RepOk function is a necessary input for Korat, as it fully represents an invariant property of a data structure in the form of a boolean-returning method located within the Java class for that same data structure.

Our team elected to use Korat to create our dataset of candidate vectors not only because of its use in the original study, but due to its effectiveness in helping us achieve every requirement of this module. An important benefit to using Korat is that it includes Java classes for many of the most commonly encountered data structures, such as Directed Graphs, Binary Trees, and Singly-Linked Lists. Furthermore, the Java classes themselves include several pre-existing RepOk methods for each of the data structures. Therefore, to create a complete dataset for many different invariants, we used the included classes within Korat and also used the existing methods as reference points in learning how to write the RepOk methods. Once our team finalized a set of graph invariants to be included in the final dataset, we wrote the RepOk methods for each invariant. Korat is based in Java, so the RepOk methods were easy to create for new data structures of increasing complexity. Furthermore, we found Korat to be the most commonly used data generation tool in several related works, which allowed our team to consult a wealth of documentation supporting Korat development, as well as a number of people with experience using Korat, who were useful to us as support throughout.

We decided on five graph invariants -- contains self-loops, K-regularity, acyclicity, density (greater than 0.70), and contains exactly one root -- for our ML models to learn to verify. For this deliverable, our team wrote five RepOk methods that each verify one of these graph invariants. Each method takes a Directed Graph, as represented by the existing DAGNode class, as an input and returns its boolean classification in regards to the invariant. Our team successfully integrated these RepOk methods within Korat, and then used Korat to generate a dataset containing valid and invalid data structures of varying sizes for each invariant.

3.1.2 Python Scripts: Parsing and Splitting

In order to use the data generated with Korat to train and test the models, our team needed to manipulate the data by parsing it. We created a Python script to parse Korat's output and generate datasets that were compatible with the ML models input format. Finally, we took the resulting dataset and split it into training and testing subsets using our splitting script.

The splitting script begins by taking, as input, Korat's command-line printed output, and parses the output as text, line by line. The raw output produced from Korat is a list of vectorizations of each data structure; i.e. each generated data structure is represented as an array (or "candidate vector") corresponding to the data structure vector representation used internally within Korat. Furthermore, if any data structure from the list is also a valid structure per the invariant, Korat will display a "***" symbol next to the valid vector. For example, two lines of the output may appear as:

```
1,0,2,0,3,0,0,0***
1,0,2,3,0,0,0,0
```

where the first line represents a tree data structure which possesses the connectedness invariant, and the second line is another tree which does not. These two lines correspond to the following train/test dataset matrices:

$$X = \begin{bmatrix} 1 & 0 & 2 & 0 & 3 & 0 & 0 & 0 \\ 1 & 0 & 2 & 3 & 0 & 0 & 0 & 0 \end{bmatrix} \quad y = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

where X and y are represented first as a Python matrix (i.e. list of lists) and list, respectively, then as Pandas Dataframes, then finally as a Numpy matrix and array, respectively. Note that Pandas and Numpy are both commonly used software packages in Python for data analysis and linear algebraic computations. Furthermore, valid vectors (i.e. the rows) in X are denoted with the target variable $y = 1$; likewise invalid vectors are denoted as $y = 0$, as shown in the

y vector above.

Using the previous X matrix and y vector as inputs, the splitting script randomly splits the data into testing and training disjoint subsets. More specifically, the script selects some percentage of the data for training the ML models (say 90%), and the remaining percentage percent (10%) for testing. The 90/10 split ratio is an optimal train/test ratio, as proven in the paper, *A Study of Learning Data Structure Invariants Using Of-the-shelf Tools* [1]. Using the `numpy.random.choice` Python library and this split as an input, the script randomly samples, with uniform probability distribution, 90% of the valid vectors for training, and the remaining 10% is then used for testing. In practice, a 90/10 split ratio is not always conceivable, typically due to efficiencies in generating data and training models. Knowing this information, we aimed to observe the efficacy of splitting the data with varying train/test ratios. In our final design solution, we used train/test ratios of both 10/90 and 50/50.

The final product for the Python scripting deliverable are the two scripts for parsing and splitting, which have been uploaded to our GitHub repository.

3.1.3 GitHub Repository with Persistent Data

The final deliverable for this module is our GitHub Repository, which stores all of our generated, parsed, and labeled training and testing data, as well as code for both the Korat RepOk methods and, in later modules, the data pre-processing and model training and testing. It was important that the data remain constant, as we used it as input to various ML models through multiple iterations of training and testing, and during the comparison of models and their outputs. Therefore, it was crucial that we saved all data used, and that the data remained untouched after the completion of this module.

3.2 Machine Learning Models

Using the datasets from the Data Generation module, we trained a set of ML models to accurately predict a vectorized data structure's validity per any given invariant in the Machine

Learning Models module. We focused on three types of ML models: Decision Tree, Support Vector Machine, and Neural Network. Though there are many different types of models which may have been chosen for this task, our team selected this narrow list for several reasons. These three models are among the most common and general types of models for ML classification. This gave us access to widely available tools and documentation for helping create, tune, and debug our models, as well as ample opportunity for comparison of our models' performance. Additionally, there exist more ML models for classification which are closely related to one of these three. For example, if our Decision Tree model performs as the best classifier of the three for a given invariant, further work might consider exploring models such as Random Forests and Gradient Boosting, which are more advanced architectures of Decision Trees. The generality of each model means any related models are likely to be more advanced, and may offer better performance. Lastly, our small set of models is diverse so each model's architecture and algorithm is significantly different from one another, as are those of any derivative models. This allowed us to maximize performative capability across all models that our team explored.

We compiled a list of the aforementioned three models as well as possible future additions in Table 1. We trained and tested the three required types of models using the datasets from the Data Generation module, and then analyzed each ML Model's results. To complete this process, we used two files that are included in our GitHub deliverable: Python notebooks of trained models and a table of ML model evaluation metrics.

Table 2. Machine Learning Models

Model Name/Type	**Library**	**Category**	**Inclusion in final report**
Decision Tree	Scikit	*Decision Tree Classifiers*	Required
Random Forest	Scikit	*Decision Tree Classifiers*	Optional

Gradient Boosting	Scikit	*Decision Tree Classifiers*	Optional
Adaboost	Scikit	*Decision Tree Classifiers*	Optional
Support Vector Machine	Scikit	*Support Vector Machines*	Required
Multi-layer perceptron	Scikit, Keras/Tensorflow	*Neural Networks*	Required
Convolutional	Keras/Tensorflow	*Neural Networks*	Optional

3.2.1 Python Notebooks of Trained ML Models

Our first deliverable for this module is the set of off-the-shelf ML models that our team built and trained using Python. Each of the models is imported from either the Scikit Python library or the TensorFlow library, as seen in Table 1. We created one Jupyter Notebook, a Python interface commonly used for collaboration, for each ML Model such that in every Notebook there is one trained model for each data structure. This model reads a data structure from the testing subset and accurately predicts whether the data structure holds the specified invariant properties. In both the Scikit and Keras machine learning frameworks, the entire list of data structure samples designated for training is passed to the model. The model tries to predict if the invariant properties hold for each sample in the list, and when its predictions are wrong, each model adjusts its internal tools and parameters that it used for that prediction. Each model has a different algorithm and approach for generating its predictions; however, the general intention of the "learning" phase produces the same result, a trained model to classify if a vectorized data structure meets the criteria of an invariant or not. Once all of the models were trained, we exported the Jupyter Notebooks as executable Python files to use in the Integrated Testing Tool module.

3.2.2 Charts of ML Model Evaluation Metrics

Our team created a series of charts of Table of Machine Learning Evaluation Metrics used to evaluate the efficacy of each of the models at predicting each data structure. The metrics we

explored to evaluate each model are listed below, and the full, expanded chart with greater detail about each metric has been provided in Table 4 in Appendix F.

- Binary Cross Entropy Loss
- True Positive
- False Positive
- True Negative
- False Negative
- Accuracy
- Precision
- Recall
- True Positive Rate
- False Positive Rate
- Receiver Operator Characteristics- Area Under Curve
- F1 Score

For each graph invariant, we computed the metrics for every trained model. This included varying train/test splits for each model. Furthermore, we also varied the distribution of valid/invalid data points in our datasets, with one data set consisting of a 50-50 split, and another consisting of the natural distribution of the invariant's data. Table 2 is an example chart (condensed for clarity) for one graph invariant. Each metric was used to understand how the models performed. Using these metrics, our team determined the best ML model types to predict the validity for each data structure, and we used the best performing ML model types in the Integrated Testing Tool module.

Table 3. Metrics to Assess ML Models (condensed)

Models Train/Test Split	**Metrics**					
	MSE	**BCEL**	**Accuracy**	**AUC**	**Precision**	**...**
Decision Tree						

10/90:						
Equal (invalid/valid)						
Random (invalid/ valid)						
50/50:						
...						
Support Vector Machine						
...						
Multi-Layer Perceptron						
...						

3.3 Integrated Testing Tool

The Integrated Testing Tool module uses the trained ML models for data structure verification to demonstrate a potential application in a runtime testing or unit testing environment. In this tool, the ML models are integrated into a Java Unit Testing suite which uses models to validate data structures in each of its tests as if the models were standard algorithms implemented in Java. Our team chose the most accurate and efficient ML model types to use based on the Chart of Machine Learning Evaluation Metrics deliverable from the ML Models module. The Integrated Testing Tool module has two deliverables: JUnit Test Files with our trained ML Models and a Chart of Runtime and Memory-Use for JUnit Tests to compare the time and memory efficiency of the tests with and without the ML Models.

3.3.1 JUnit Test Files

Our first deliverable, the JUnit test files, verify complex data structures in a Java program at runtime. Our team chose to use JUnit because it is a widely used testing suite in industry, it is written in a popular language (Java), and the content and execution of the program being tested by JUnit tests is arbitrary as long as the program is written in Java. In order to use the trained ML models from the ML Model module, we exported the models from the Python notebooks as executable Python files and added the files to the testing directory.

We wrote one JUnit test file for each invariant/model pair. Each file consists of a set of testing methods and the exported model wrapped in a Java method. Each testing method creates an example of a data structure that either does or does not meet the criteria for the given invariant and then confirms that the model has properly classified that data structure. Additionally, we included a Graph class for building the example graphs and a method that converts an instance of the Graph class into a candidate vector so it can be used as an input to the model.

3.3.2 Chart of Runtime and Memory-Use for JUnit Tests

Additionally for the Integrated Testing Tool module, our team created a chart containing time and space complexity metrics for the JUnit tests using the ML Models. We collected the memory and time use for each JUnit test method running within the JUnit test files. While we had previously hoped to compare our metrics to standard Java testing methods, we determined that this comparison would be trivial due to the limitations of our models. Standard methods, such as recursive or brute force methods, have complexities that increase with the size of the graph being tested. Our models are trained on small graphs (i.e. less than 7 nodes), and therefore our time and space complexities are very small. Although we did not directly compare our results to standard methods, we did record and analyze our results in a chart (template shown in Table 3), as well as make recommendations for future extensions of our project.

Table 4. Template for Chart of Runtime and Memory-Use for JUnit Tests

Invariant	Model	Execution Time	Memory Use

3.4 Final Paper

This Final Paper, the last module of our design solution, summarizes and analyzes the results of our entire system. Because our project is research-based, it is important to thoroughly document the process of our systems so that future researchers may reproduce, and hopefully extend, our results. Therefore, we made sure to include the data generation methodology, ML model generation methodology, and process for building the Integrated Testing Tool in this final paper. Additionally, we will use this paper to communicate our final evaluation of the potential efficacy of ML data structure verification in a testing tool. The ML Model Evaluation Metric Chart from the ML Models module and the Chart of Runtime and Memory-Use for JUnit Tests from the Integrated Testing Tool module will serve as the data to support the final evaluation.

4.0 DESIGN IMPLEMENTATION

With our system design solution thoroughly crafted, our team began to delegate and complete the implementation of our system. Throughout our implementation, our team encountered a number of obstacles which continuously challenged certain expectations of the project going forward. We discuss all the challenges and their results for each of the system modules.

4.1 Modified Invariants

In the early phases of our project, we changed the graph invariants from complex properties (e.g. bipartiteness, identifying a tree, planar) to simpler properties like “does the graph contain a cycle?”. The complicated invariants contain significantly fewer valid graphs as compared to less stringent invariants. So, the ML models could not accurately learn the complex invariants because there were not enough graphs that met the invariants in the Korat data sets. By contrast, the simpler invariants contained thousands of valid graphs in the data. Our team decided to train the ML models on simpler and crucial invariants -- contains self-loops, K-regularity, acyclicity,

density (greater than 0.70), and contains exactly one root -- so we would not have to sacrifice the accuracy of our ML models.

4.1.1 Problem

Although our initial design proposed testing for complex graph properties such as planarity and bipartiteness, we discovered that the data sets for complicated invariants did not contain enough valid graphs to accurately train ML models. Our team was excited to automate the validation of sophisticated graph properties with our ML models because many algorithms rely on complex graph structures. However, after writing the appropriate RepOk methods for these properties and integrating them into Korat, we found that the datasets were large but contained very few valid samples. For example, one invariant we tested was checking if a graph was a tree. For a graph of 6 nodes, there exist only 20 valid trees because trees have very strict definitions. With only 20 valid trees, training an ML model would be pointless because it would be easier to just manually check if the graph was one of the few possible trees. Although we could have increased the number of nodes on the graphs to get more valid possibilities, the larger graphs produced exponentially more invalid graphs and our personal machines were not able to produce data sets for graphs larger than four to five nodes. So, our group had to reconsider the complexity of the invariants we wanted to test.

4.1.2 Solution

Although complex invariants produced invaluable data sets, our team discovered that by breaking down the complicated invariants into simpler ones we could create beneficial data sets and use multiple ML models to check for more stringent properties. Some complex invariants can be expressed as the conjunction of many simple invariants. So, difficult graph properties can be tested by running the graph through each model for the simple invariants. If all models return true, the complex invariant holds for the given graph. For example, one invariant of a regular tree is that it can only contain one root. So, we developed an ML model that automatically checks if a graph has one root. If the output of the ML model checking the one root invariant is false, then we can assume that the graph is not a regular tree, which is useful information for many software

engineering problems. Our team also improved the accuracy of our ML models by using uncomplicated invariants because the simpler properties allowed for more positive/valid samples in the output from the data generation phase of our project. Given more positive samples, we could more accurately train the ML models, which led to more promising metrics for the models such as accuracy, precision, and false positive rate. Thus, our team developed a solution that allowed us to still check for some complex invariants and create valuable data sets for ML model training and testing.

4.2 Modified ML Model Types

Our team encountered a number of challenges during the completion of the ML Models module. The results of these encounters were changes to our project goals and schedule for the rest of this module's implementation. One example is our team's initial intention of using a wider range of ML models. However, as we implemented the ML Models module, it became clear we would need a smaller, yet still representative list of models with which to train using the invariants.

4.2.1 Problem

Initially, we wanted to cover a longer list of ML models. However, our team eventually concluded that, for the scope of this project, it was unfeasible to train every model we initially considered. Furthermore, not all of the models were well suited towards the nature of this project's data structure data, like Convolutional Neural Networks or Simple Regression. The full list of models initially considered is shown in Table 2 in section 3.2.

An issue that our team encountered during model training was related to performance of the Support Vector Machine (SVM) model. Given the nature of the SVM's hyperplane-based learning algorithm, the SVM suffers from longer training times and larger memory requirements than the other included models.

Finally, due to the invariants being easily learned by the ML Models, we decided to devote less project resources to training and tuning individual models. We were able to train our models on a

large number of datasets and with a variety of train/test splits, so this was not necessarily a problem. Nevertheless, it involuntarily caused an unexpected redistribution in the allocation of our team's resources such as time and computing power.

4.2.2 Solution

Our team's need for a reduced set of ML models led us to examine several possible solutions. We ended up focusing on three specific models: Decision Tree, SVM, and Neural Network. This decision was motivated by the wide use of these specific models in industry. These models are also greatly representative of their respective algorithm classes, so discluding some models did not completely remove those learning algorithms' characteristics from the training set.

Another change we made in our implementation was that we did not tune the hyperparameters of our models. Originally, we had planned on tuning the hyperparameters of the models for better accuracy. However, the untuned versions of these models ended up being more than enough to train on the different invariants, as many of the graph properties were easily learnable by the ML models. In deciding not to attempt hyperparameter tuning, we instead focused on running more tests on our RepOk methods, the Integrated Testing Tool module, and the full system end to end. We spent more time on training more models with the increasingly abundant data being produced by Korat. We spent much of our tuning/testing time on a variety of datasets and trials. For one model and one invariant, we ran three trials per each of the four different data configurations. This process resulted in 12 test runs per model, and totaled 36 for each invariant.

We also experimented more with feature engineering in efforts to study how the models performed differently for each new set of features. One example is training models with the graph representation as vectors directly from Korat, or with graphs represented more typically as adjacency matrices.

As previously mentioned, SVM in general takes longer to train than many other off-the-shelf models. Our team found that to be true in our experience implementing this module, where SVM

suffers from a factorial order of input space complexity from Korat. Our team was forced to drastically limit our training capacities for the SVM in some cases, and in others we had to forego the training of the SVM altogether. This phenomena can be viewed in Appendix A, which contains the full table of ML model metrics. We decreased our train percent for SVM to as little as 1% of the data for very large datasets, and this still took very long to complete. There were several instances in which we were unable to run the SVM at all. Therefore, our results and models used in the Integrated Testing Tool both heavily rely on the other two model types.

4.3 Modified Dataset Properties

In order to improve the efficiency and accuracy of the ML models, we made two changes in how we produced the data used to train the models. First, we reduced the size of the training data for SVM models to decrease the time it took to train an SVM model. Second, we created training data that reflected the original distribution of valid to invalid vectors produced by Korat to closer replicate realistic data.

4.3.1 Problem

We were motivated to change our training data because we recognized two significant problems with the efficiency and accuracy of our ML models. The first problem regarded the efficiency of training the SVM Models. When we tried to train SVM Models on the same size datasets used to train the other models, the model would take hours to train or cause the computer to crash. Regarding the second problem with the ML model's accuracy, we questioned if the models could be accurate with real data that naturally has a higher percentage of invalid vectors, since we were originally only training the models on data that had a 50/50 split of valid and invalid vectors.

4.3.2 Solution

In order to solve the problems with the efficiency and accuracy of the ML models, our team made two changes to the parameters of the training data. To increase the efficiency of training an SVM Model, we decided to reduce the size of the training dataset for SVMs. Thus, we trained the SVM data set with a 10/90 and a 1/99 train/test split; in contrast to the other models that are

trained on 10/90 and 50/50 train/test splits. This is why we were able to train the SVMs with different train/test splits with little cost to the accuracy of the models. We conduct an analysis of potential alternatives to this solution in Section 8.1.1 Creating Larger Datasets. To potentially increase the accuracy of the models, we replicated a training set on real data by using a natural random distribution of valid and invalid vectors. Our team decided to compare the ML models trained with balanced versus a random invalid/valid split. In many, but not all, invariants, the random distribution yielded a more accurate model.

5.0 TEST AND EVALUATION

Our team created and completed a series of methods to analyze the success of our system design and implementation. Like our approach to other aspects of this project, we have divided our testing and evaluation goals and strategies between each of our system's three technical modules. In this section, we describe our procedures and discuss the quantitative results, achievements, and shortcomings of our implementation for each module.

5.1 Data Generation Module

Testing and evaluation of the Data Generation module primarily focused on Korat. Our team first verified the initial utility of Korat, then tested and validated our invariants and RepOk methods created for Korat, and finally inspected the new data generated from Korat for accuracy, all while completing and testing necessary alterations to our project's code base.

5.1.1 Method

Since our testing process for this module was centered around Korat, it is worth noting that Korat has itself already undergone significant testing and evaluation prior to our usage of it. The original creators of it as well as other researchers have extensively used Korat for over a decade. Additionally, Korat's creators created an extensive testing suite containing tests which cover all aspects and details of Korat's functionality. Thus, our team focused our testing efforts of the Data Generation module on aspects of Korat which were important for our needs and goals.

First, our team aimed to understand and mitigate the difficulties that could unexpectedly arise while beginning to use and operate Korat. These difficulties include those specifically encountered during the first stages of our usage of Korat: installing the Korat library on machines, resolving any missing Java dependencies, assessing compatibility with our Java platforms, other system modules and components, and outstanding machine compatibility issues such as operating system requirements. Our ultimate goal was to achieve basic, active functionality of Korat before attempting any further analysis or modifications.

Once we finally were able to use Korat, we began implementing the finalized collection of invariants. Testing our new invariants involved an extensive code review by at least two team members per invariant, as well as writing many unit tests to verify their accuracy. During the process of writing the RepOk methods, we discovered a few logical errors in the Korat library as well as areas which did not satisfy our system design requirements. Thus, our team would have to modify parts of the pre-written library code and then test them similar to our testing of newly written code.

Finally, after making the necessary additions to the Korat codebase, our team had to evaluate the validity of the new data we could then generate. The highly technical and complex nature of how Korat explores and creates candidate vectors made testing the vectors' validity tricky. We had to trust that Korat's internal library code had been sufficiently tested, and were only able to further verify this by hand-inspecting a small, random few of the sometimes millions of vectors it produced. To aid in this inspection, we made use of a tool in Python which can take the vector representation of any directed graph and visualize it as a picture of circles for nodes and arrows for edges. We were then able to inspect the picture to verify that its structure agreed with that of its corresponding vector, as well as gain a better understanding of how Korat algorithmically explores the space of all possible candidate vectors.

5.1.2 Results

The first goal of testing the Data Generation module was to achieve, and understand how to achieve, basic functionality of Korat across all users and systems. The result of this was successfully installing and demoing usage of Korat on both the Windows and MacOS operating systems. MacOS proved to be easier and faster, and facilitates an easier extension of Korat to Linux OS. There were challenges with installing and usage with Windows, which included proper setup and verification of system environment variables, difficulty installing and resolving the dependent libraries needed for Korat, and a lack of Unix-style development tools and practices such as a bash terminal. Another challenge we resolved was the Java-specific dependencies and requirements of Korat. Our team discovered that Korat requires usage with Java 1.5, whereas most of our team's machines were configured for use with Java 1.7 or 1.8. We were able to install and revert back to the older version of Java, as well as restrict our future development within Korat to the restrictions of Java 1.5.

Next, our team began writing, reviewing, and testing our RepOk methods for our five chosen invariants. This amounted to thoroughly reviewing at least 200 lines of code, as well as modifications to the code within Korat. Our team created five JUnit tests for each of our RepOk methods. All of the tests passed, and we were able to make use of these JUnit tests again for the Integrated Testing Tool module later on.

With the RepOk methods successfully written and evaluated, our team began generating the candidate vectors for each invariant using Korat. Though Korat successfully generated the data thanks to our previous library installation and testing, we felt we had to verify the accuracy of the output data. To achieve this, we found a method to visualize any Korat-produced candidate vector. We used this method so that we could select a sample of vectors and visually inspect them for correctness. Below is an example of graph visualizations.

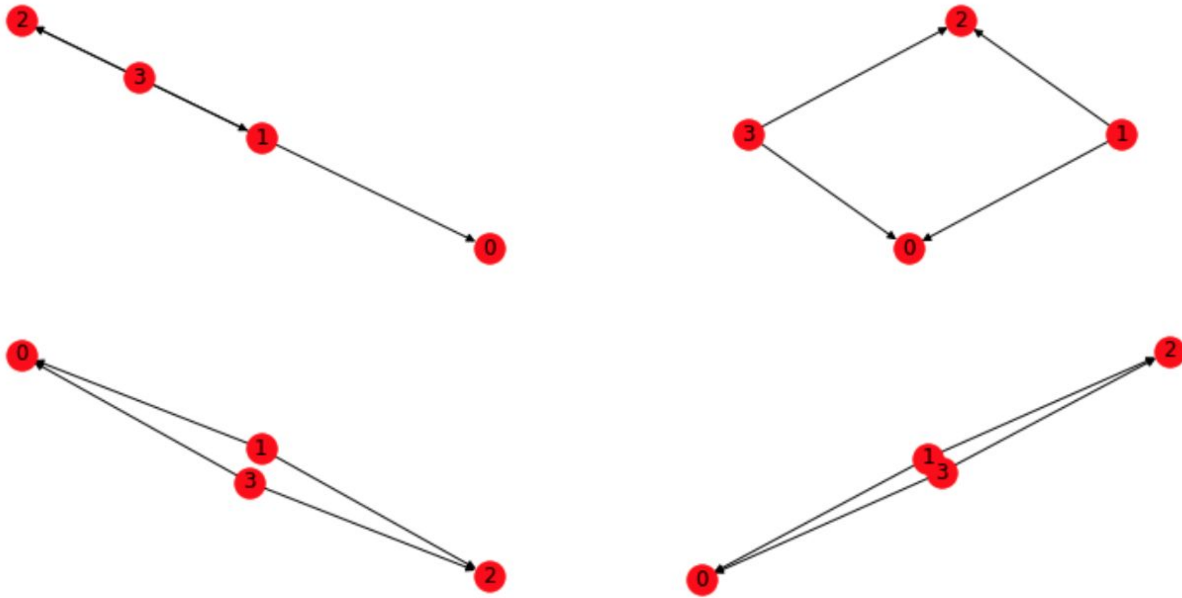

Figure 2. Visualizations of Korat candidate vectors

We gathered a few important details about Korat while inspecting these visualizations. First was an error in Korat's code which only allowed each node in a size-*n* graph to have a maximum of *n-1* nodes. Second, the visualizations do not clearly show two features: repeated edges, and self-loops. This drew to our attention the fact that Korat could have repeated edges at all, something which not only is not common in real-world examples of graphs, but dramatically increases the runtime and memory of both Korat and model training.

Finally, we evaluated the performance of Korat for each of our RepOk methods, as well as the memory and time metrics for Korat's candidate vector generation.

Table 5. Korat Runtime Metrics per Invariant

Invariant	Graph size	Actual # valid from Korat	Total graphs explored by Korat	Time taken to generate graphs (sec.)
Self-Loops	3	22	75	0.199

	4	3952	19402	5.471
	5	651026	5235051	2397.863
Acyclicity	5	4858	35464	8.176
Regularity	4	100000	15246455	N/A
Density	7	104268	2097152	1214.877
Single Root Node	3	1624	10699	2.037

As shown in the table, as the size of a graph increases, there is a sharp increase in both the number of candidate vectors and runtime of Korat. Furthermore, the increases can be very high both for very small graph sizes and between any successive sizes. This is attributed to the exponential, and often factorial, orders of the formulas which model the runtime and memory metrics of Korat. The time and memory complexities of Korat's vector exploration increase rapidly and without bound.

5.1.3 Analysis

Testing the Data Generation module overall allowed our team to identify and correct errors in our code implementation, reach better informed and quality design decisions, and ensure correct output from this module for use in the next. We created several useful tools that we then used for building and testing in our next two modules. Specifically, the method of visualizing graphs was useful for Feature Engineering in the ML Models module, and a greater understanding of Korat's graph exploration algorithm helped us redesign our choices and implementations of invariants and subsequent training methods. We were also able to use modified versions of the JUnit tests in the Integrated Testing Tool module.

5.2 ML Models Module

Testing and evaluation for the Machine Learning Models module consisted of testing the data flow process with a smaller data set to full-fledged data collection to evaluate the machine

learning models themselves. In general, since the ML models are already tested/debugged as a part of their respective libraries, the focus of this section is more on our metrics/evaluation.

5.2.1 Method

Since our project used off-the-shelf ML models, we did not need to focus on the functionality of the models like we did in Korat, but rather we aimed to test and evaluate the accuracy of our trained models. Our initial goal for this module was to complete an end-to-end process, so we first tested the models on a smaller sample. The code we used for this initial run formed the basis for the rest of our machine learning models and metrics collection. From an evaluation standpoint, we needed to focus on how the models learn the invariants of our data. For this, we had a set of metrics (mentioned earlier) that we evaluated the models on. We ran three trials for each configuration and took the average for each of our listed metrics.

5.2.2 Results

In general, our results consisted of high accuracy and precision. Our full collection of results are contained within the tables in Appendix A. These results showed us that most of these invariants are easily learned by the machine learning models. When training on the engineered 50-50 data sets (labeled as Balanced) compared to the original distribution of valid/invalid graphs, we saw the 50-50 datasets often had an advantage in training. For instance, a Decision Tree model would be able to dedicate more of its pruning to isolating the positive cases when it does not have all of the negative cases to cipher through. In summary, when training on datasets with the distribution of valid/invalid determined by Korat, we ended up with more realistic results, albeit with slightly worse metrics compared to the balanced datasets.

5.2.3 Analysis

The goal of our analysis was to select one model per invariant to be used in the Integrated Testing Tool module. The main metric we referred to was the AUC-ROC score, because it adjusts for the distribution of data points. For example, if we are testing on data which is 90% negative, it would not make sense to give a score of 90% if a model is simply able to predict

negative for every data point. Hence, in general, we looked at the AUC-ROC score, but we also tried to have some diversity in the models we chose for each invariant, which is why we ended up choosing mostly Decision Tree models but also SVM and Neural Network.

5.3 Integrated Testing Tool Module

The Integrated Testing Tool module integrates the Data Generation module and the ML Models module to create an application that could be used outside of a research project. To determine whether or not we were successful in implementing the design of this module, we used various testing methods, including static analysis and JUnit tests.

5.3.1 Method

In order to create a test suite that mimics a real-world application, we exported our models into a Java project and added functionalities that would make them compatible with the Java syntax. To start this process, we opted to use adjacency matrices to represent the inputs to our test methods by creating a Graph class that implements an adjacency matrix to represent a directed graph. Then, we wrote a Java method that converts an instance of the Graph class into a candidate vector. This method is the first line of the method that wraps the exported model, so the model can recognize the format of the input. To test the conversion method, we created several example Graph objects and made sure that the output candidate vector depicted an identical graph. Next, we began populating our test suite with test methods. We created 5 classes, one for each graph invariant, and wrote around 6 test methods in each class. Each method creates a unique instance of the Graph class, sends it as a parameter to the trained model's wrapper method, and then asserts that the model returns the correct boolean classification of the graph. We used static analysis and peer code review to verify that our test methods asserted the correct classification of the example graphs. Finally, we imported the models into our Java project. Since this module is fundamentally a test suite, once the models were imported, running the test suite was an effective and efficient way to test the functionality of the model within the wrapper method.

To get a better understanding of the success of our module, we measured the time and space cost of each model within their Java wrapper methods. To measure the execution time of each model, we iteratively ran the model with each graph input 5 times and then calculated the average time of the 5 iterations. We determined the space of each model by its file size.

5.3.2 Results

We successfully implemented a test suite that uses 5 different trained ML models to classify graphs based on 5 different graph invariants. This process included creating a method to convert an adjacency matrix into a candidate vector and test methods that exhibit the correctness of the imported model. We were also able to measure the time and space costs for testing each invariant with respect to their trained model. Table 6 shows the completed Chart of Runtime and Memory-Use for JUnit Tests.

Table 6. Chart of Runtime and Memory-Use for JUnit Tests

Invariant	Model: Data distribution, train/test split	Execution Time	Memory Use
Self-Loops	Decision Tree: Balanced, 10/90	36 ms	7 KB
HasOneRoot	Decision Tree: Full, 10/90	31.2 ms	12 KB
isRegular	Support Vector Machine: Balanced, 10/90	209 ms	1.2 MB
isDense	Neural Network: Full, 10/90	150 ms	256 KB
isAcyclic	Decision Tree: Full, 50/50	70.5 ms	130 KB

5.3.3 Analysis

We were able to meet specifications for this module. The time and space costs of using our models are feasible for a real-world application. Nevertheless, this could be attributed to the fact that our graph sizes are very small. With larger graphs, the size of the ML model may increase significantly. Since our team could not produce larger graphs due to the limitations of our

machines, we could not test how the ML models increased in size in relation to increasing graph sizes. Further, with larger graph sizes, the program will take a longer time to convert the graphs into a Korat vector. The function that converts the adjacency matrix into a Korat vector has a runtime of O(|V| * |E|), meaning the time to execute the function can potentially increase polynomially with respect to increases in the graph size input. Although the Integrated Testing Tool module met specifications on the surface, the implementation needs to be tested and refined further before it can be a useful tool in industry. In testing the Integrated Testing Tool module, we were primarily concerned with testing the efficiency of the ML models and not the accuracy. However, in Section 8.2.3, we present that the accuracy of the ML models is much lower with randomly generated data structures, instead of data structures produced by Korat. The false confidence on our ML models proves that further research needs to be done to develop more accurate models when dealing with real-world data before the ML models can be implemented in a testing tool.

6.0 TIME AND COST CONSIDERATIONS

Throughout the implementation of our project, there were points where we had to modify our project based on time or cost constraints. Our group made schedule adjustments early in the project to accommodate for two factors: trial and error time and the long time required to generate data. However, we did not have to adjust the project for any cost considerations, given that the research is completely software-based and that we ran the software on our personal machines. Therefore, we were able to complete our project in our budget of $0 and within our decided schedule.

Initially, we had planned our schedule to sequentially work on the three modules, i.e., complete the Data Generation module for all of our invariants before moving onto the ML Models module, and so forth. However, when we discovered that the largest time constraint of our system was the data generation, we decided to only generate one invariant, self-loops, to implement and test the rest of our system design. By de-risking our entire system using the self-loops invariant, we were able to quickly identify and solve problems for every module.

Additionally, due to COVID-19 disrupting our planned schedule, our team slightly cut the scope of our project to exclude finding a real-world application for the Integrated Testing Tool module. The final step after creating the testing module would have been to make the testing suite into a package that we could implement into some sort of research or industry work. However, since our team lost a week and had to adjust to a new work-from-home schedule, we decided to cut the research/industry application. Instead, we created an arbitrary program that generated random graphs in the form of adjacency matrices, a commonly used data structure, and integrated the ML models to evaluate if the graphs met an invariant.

It is worth noting that resources devoted to training ML models often do incur a financial cost. Many similar projects use a distributed computing approach where the training occurs over a server hosted by a company like Paperspace or Google Colab for a fee. This is done in order to access more powerful hardware for model training and, thus, decrease the overall train time. However, our team was able to complete our project by training the models on our own personal machines, which had enough computational power to complete the training within a reasonable time.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

Although our system design is abstract and research-focused, our implementation can be integrated and extended in future research, which could potentially lead to safety and/or ethical concerns. In fact, in the Integrated Testing Tool module, we created a testing methodology that can be easily applied to both industry and research projects. Therefore, our team has an ethical responsibility to truthfully evaluate any bias in the Data Generation module, the accuracy of the ML models, and any safety concerns for implementing the Integrated Testing Tool into a real-world system.

Through our research, we discovered that Korat isomorphically prunes the resulting data set, causing a bias in the Data Generation Module. Since Korat was created as a test-input generation tool and not to generate data for ML, the implicit biases of using Korat as a data-generation tool

are not well defined. So, any team using ML models trained with Korat data needs to be cognizant that the biases can negatively affect the performance of the ML models in applications with real data. Still, Korat impressively explores the state space for data structures. So, as proved by our research, the test-input generation tool will be useful for further research in training ML models to verify data. However, alternative methodologies or changes to Korat may have to be implemented to reduce the bias of the data.

In the ML Models module, our team meticulously measured the accuracy metrics of the ML models and extensively reported our evaluation process to preserve the integrity of the research. We implemented common scientific research methodologies like averaging the results of three trials for each metric listed in Appendix A. Table of ML Model Metrics Results. Further, we described in detail the process and tools used to test the ML models in Section 5.2 ML Models Module. So, any future teams that wish to replicate or extend our research can interpret our results correctly.

Finally, real-world systems making use of our Integrated Testing Tool module pose a safety concern because our system does not guarantee that the model will make an accurate prediction. Users should be aware of the risk of models not accurately predicting the validity of every possible input. Though the models successfully predict the majority of inputs, rarely do models predict every input, and thus there are a number of false positives and false negatives. Therefore, any systems that are critically dependent on accurate predictions at runtime should be aware of the false positives and negatives, and should use the regular RepOk method for validation at runtime.

8.0 RECOMMENDATIONS

In this section, we describe future extensions to our project and the alternative design decisions we could have made during the course of our project. Many of the future extensions rely on greater computational abilities, a service that was not readily available to us during our

implementation. Similarly, the alternative design decisions, while theoretically feasible, were not explored simply due to the time constraint of completion within a semester.

8.1 Recommendations for Future Extensions

The four recommendations we propose for groups trying to extend our work are in the areas of improving the base datasets, improving Korat's implementation scheme, modifying the models, and finding additional applications.

8.1.1 Creating Larger Datasets

An idea for a further work is to generate larger data sets that lead to more well-trained machine learning models with the aid of supercomputers, and to explore different data generation techniques such as those based on model checking or symbolic execution. For our project, we used Korat as our data generation tool. Korat generates samples of a data structure subject to constraints provided in the RepOk method. We limited the size of our datasets on the order of a million samples since our personal computers could not handle much more due to both memory and time limitations. With access to a supercomputer, a researcher could generate datasets with billions of samples. A supercomputer would be able to not only generate these datasets quickly, but also train ML models on them quickly.

Another stream of exploration could be in different data generation methods entirely. Methods based on model checking or symbolic execution may lead to different and interesting results. In short, Korat's highly specific method of producing candidate vectors can greatly reduce the accuracy of models used in real-world applications. Therefore, other data generation techniques that output data with more robust algorithms may lead to different results when it comes to model accuracy and prediction metrics.

8.1.2 Making Korat Improvements

With more time to modify Korat, groups interested in further extensions should consider building new base data type classes that can easily implement other types of graphs, such as undirected

graphs, weighted graphs, randomly generated graphs, and hypergraphs. For our usage of Korat, we essentially treated it as a black-box; using it as is, without modifying its existing classes/data structures. We were able to write RepOk methods built around the basic data type classes already implemented in Korat. For instance, we were able to implement directed graphs in Korat using the pre-existing DAGNode class. Despite its name, this class can represent all directed graphs, not just Directed Acyclic Graphs (DAGs). However, we were unable to truly implement undirected graphs. In theory, we could have built an undirected graph class using DAGNodes by ensuring that for every edge from node U to node V, there was an opposite facing edge from node V to node U. However, this would have made analysis and implementation of undirected graphs complicated. So, improved or altogether new base data classes in Korat could lead to interesting results for more advanced graph types.

8.1.3 Modifying ML Models

In our study, we wanted to test the learnability of graph properties by machine learning models that were not tuned to do so. Although we were able to collect positive results, perhaps model accuracy could be improved by hyperparameter tuning these models as well as data transformation methods such as feature engineering. Improvements like this could be crucial if our tool were to be deployed in a setting where there is little tolerance for error, such as the healthcare or transportation industries.

8.1.4 Finding Additional Applications

Checking program and software correctness is an important research direction for the computer science community. That being said, finding large real-world systems that could benefit from automated program correctness tools, such as the one we developed, would provide further motivation for this stream of work. Some existing methods struggle with verifying complex properties such as the graph properties explored in this paper. Other works that follow our workflow have not been applied towards graph properties. Our work could have applications in automated program analysis tools as well as large software systems that are widespread in

industry. Given our compressed timeline, we were unable to find specific projects that use work similar to ours.

8.2 Alternative Design and Implementation Decisions

We concluded three key considerations for system redesigns or changes to our implementation process which would have either improved our project's outcome or better equip our system for future use and research. These include: improving the model training and tuning process, training the models with libraries which are not only dependent on Python or any single language or system architecture, and evaluating biases within the domain of our project's requirements, specifically Korat.

8.2.1 Streamlining Model Creation

A streamlined workflow to take a property through the data generation, machine learning, and integrated testing phase would have allowed us to scale up to include perhaps tens or hundreds of invariants. Though our team was successful in completing our ML Models module, there were a few challenges which both prevented us from achieving completion faster and possibly inhibited an improved system design. In particular, the time which it took to go from generating candidate vectors in Korat to exporting fully trained ML models was a rather lengthy and cumbersome process. The narrow scope of our design goals meant our team was able to afford a somewhat inefficient model training process. However, if the number of inputs from the Data Generation module, including data structures or invariants, ever increased, a more streamlined approach would be necessary. This would essentially be a new system design which would include: automatic creation (and recreation) of every type of ML model included within the scope, automatic feature engineering and included data transformations, and automatic training, testing with metrics reporting, and selection of the best performing models per invariant. These changes would greatly increase the ability to output more ML models and iterate through the entire ML Models module as necessary.

8.2.2 Make Model Training Platform Independent (Java)

A redesign consideration that originated from the Integrated Testing Tool module was the idea of creating and using the trained ML models without the restriction of any environment or platform. We considered three implementation choices of this module. The first option relied too heavily on a user's operating and file systems. For the second choice, though it averts the system reliance problem by using Java, it does not allow for this project to be easily used with non-Java software development environments. Many developers write entire systems in multiple languages which do not include Java, and even though the idea of JUnit testing frameworks is a staple of the Java platform, similar testing frameworks and requirements exist for many other languages. Therefore, our integrated tests would have to be re-written and re-compiled for the languages of any projects which do not use Java. A third alternative implementation choice would have been to host the functionality of all three modules on a server. Users could then run integrated tests by accessing our project's code through an API REST endpoint, which is standard for many real-world systems, and would permit functionality regardless of choices of operating system or development language and environment.

8.2.3 New model evaluation and re-training methods

We discovered that though Korat is thorough in how it explores and isomorphically prunes the space of all possible graphs to explore, the isomorphic nature of its exploration actually limits the ability for a model trained from Korat's data to perform well.

Take, for example, the *hasOneRoot* invariant. Korat method of exploring all possible graphs with this property creates a set of valid graphs where the graphs, though are all pairwise isomorphic, all only have its last node set as the root node. In the case of a graph of size 4, the "3" node is always the root node. You can see an example of such a graph in the following diagram.

Change in node ordering:

[0, 1, 2, 3] → [1, 3, 2, 0]

Change in candidate vector:

[0 0 1 2 3 1 0 0 0 0 4 1 0 2 2 4 2 2 2 2 4 2 1 2 2] → [0 0 1 2 3 4 2 3 2 2 1 1 0 0 0 4 2 2 2 2 4 3 1 2 2]

Figure 3. Diagram of isomorphic reassignment of nodes in a graph

The graph visualized on the left satisfies the *hasOneRoot* invariant, with node "3" as the root node. Our team took all of the candidate vector graphs from Korat, and randomly reassigned the "names" or orderings of their nodes. In the example above, you can see how the nodes were reassigned: the "3" node becomes "0", which becomes the new root node. Note that the resulting graph is isomorphic to the original. This scenario, in which the orderings of a graph's nodes is not known, is extremely common in the real world. However, in one case when we took a model trained to high accuracy on Korat's output and tested it on the reassigned vectors, its predictive accuracy fell from 99.9% to 83%. This shows that the models, as they currently are, pose a safety risk if used for integrated tests in a real-world system. Therefore, the models should be trained on randomly reassigned output from Korat in order to greatly increase their predictive accuracy.

9.0 CONCLUSION

In conclusion, our team successfully accomplished our two main goals: determine the learnability of graph invariants with ML models and employ ML models to a software testing strategy. To answer the question posed by the first goal– "Are graph invariants learnable by ML Models?" –, we created the Data Generation module and ML Models module using the methodologies from *A Study in Learning Invariants with Machine Learning*. In the Data Generation and ML Models modules, we trained, tested, and evaluated the ML Models using data from Korat. The positive evaluation metrics for the ML Models indicate that graph invariants can be successfully learned by ML Models; thus, answering the question of the first goal. To fulfill the second goal of applying the ML models to a testing suite, we integrated the ML models into a JUnit testing suite in the Integrated Testing Tool module. The models in the JUnit test were lightweight and ran in trivial time with 100% accuracy on graphs created by

Korat. So, our system design satisfactorily proved the learnability of graph invariants by ML models and demonstrated a working application of the ML Models in a popular testing suite.

Since our system design was built under the assumption that Korat data is unbiased and under the restriction of our computing power, further work must be done to evaluate and improve the system design with randomly produced data and larger graphs. Although our system performed nearly flawlessly when tested with data from Korat, our team discovered that the models performed significantly worse with graphs produced randomly. Due to time limitations, we were not able to explore if tuning the ML models trained on Korat data could improve their performance with randomly produced graphs. Although we measured that the time and memory usage of the ML Models was trivial in the evaluation of the Integrated Testing Tool module, we could not determine if the usage statistics would still be inconsequential when the graphs increased in size. We could not determine the time and memory costs of larger graphs, because our personal machines could not create models for graphs any larger than four to five nodes. So, our design implementation leaves two significant questions to be answered in future research on developing ML Models from Korat data. Can the ML Models trained from Korat data be tuned to have better accuracy on randomly produced graphs? And, will the ML Models for graphs larger than 4 to 5 nodes be efficient in a standard testing environment? We hope that our efforts will be useful for researchers as they set out to answer these questions.

REFERENCES

[1]*Usman M., Wang W., Wang K., Yelen C., Dini N., Khurshid S. (2019) A Study of Learning Data Structure Invariants Using Off-the-shelf Tools. In: Biondi F., Given-Wilson T., Legay A. (eds) Model Checking Software. SPIN 2019. Lecture Notes in Computer Science, vol 11636. Springer, Cham*

[2] *Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Frias. 2019. Training binary classifiers as data structure invariants. In Proceedings of the 41st International Conference on Software Engineering (ICSE '19). IEEE Press, 759–770. DOI:https://doi.org/10.1109/ICSE.2019.00084*

APPENDIX A – TABLE OF ML MODEL METRICS RESULTS

APPENDIX A – TABLE OF ML MODEL METRICS RESULTS

Graph Invariant:	**Self Loops**											
Data Structure:	Directed graph with 5 nodes		*note: each value is an average of 3 model trains/tests									
Valid if:	Graph contains at least one node with an edge directed at itself											
Model						**Metrics**						
Valid/Invalid, Train/Test	BCEL	True Positive	False Positive	True Negative	False Negative	Accuracy	Precision	Recall	True Positive Rat	False Positive Rate	Area Under Curve	F1 Score
Decision Tree												
Balanced, 10/90**	0.000038376418	449874	0	450125	1	0.9999988889	1	0.9999977772	0.9999977772	0	0.9999988886	0.9999988886
Balanced, 50/50	0.000046051701	249772.6667	0	250226.6667	0.6666666667	0.9999986667	1	0.9999973316	0.9999973309	0	0.9999986658	0.9999986658
Full, 10/90	0.000012136805	450117	0	2395670	1	0.9999996486	1	0.9999977784	0.9999977784	0	0.9999988892	0.9999988892
Full, 50/50	0.000021846255	250442.3333	0	1330549.667	1	0.9999993675	1	0.9999960071	0.9999960071	0	0.9999980035	0.9999980035
Support Vector Machine												
Balanced, 10/90	0.004400552223	449894	63	449990	51	0.9998725926	0.9998592466	0.9998859167	0.9998599866	0.00014001337	0.9998725978	0.9998725792
Balanced, 1/99	0.0179557223	494786	260	494698	254	0.9994801347	0.9994742083	0.9994862691	0.9994747963	0.00052520371	0.9994801553	0.9994801415
Full, 1/99	0.028407481	492413	6	2635379	2568	0.9991775192	0.9999877714	0.9948105489	0.9999878153	0.00001218474	0.9974041361	0.9973895511
Full, 10/90	0.004887088179	449460	3	2395925	399	0.9998585043	0.9999918419	0.9991131233	0.9999933254	0.00000667463	0.9995557965	0.9995522697
Multi-Layer Perceptron												
Balanced, 10/90	0.007506071193	585951	115	585641.3333	139.6666667	0.9997826793	0.9998038223	0.9997616883	0.9997616883	0.000196348035	??	0.9997827461
Balanced, 50/50	0.004633304043	325726	18.66666667	325212.6667	68.66666667	0.9998658528	0.9999427172	0.9997892255	0.9997892255	5.74E-05	??	0.9998659613
Full, 10/90	0.01042432736	585298.3333	697.6666667	4124825.667	724.3333333	0.9996981882	0.9988108945	0.9987640366	0.000169115070	0.9987640366	??	0.9987872072
Full, 50/50	0.000325483509	325342	6.333333333	2292159.333	18.33333333	0.9999905763	0.9999805307	0.9999436581	0.9999436581	2.76E-06	??	0.9999620938

Graph Invariant:	**Has One Root**											
Data Structure:	Directed Graph size 4 nodes											
Valid if:	The (connected) graph contains only one node with only outgoing edges											
Model					**Metrics**							
Valid/Invalid, Train/Test	BCEL	True Positive	False Positive	True Negative	False Negative	Accuracy	Precision	Recall	True Positive Rate	False Positive Rate	Area Under Curve	F1 Score
Decision Tree												
Balanced, 10/90	0.003457819854	788936	62	789119	96	0.9998998868	0.9999214216	0.9998783398	0.9998783398	0.0000785662383	0.9998998868	0.9998998799
Balanced, 50/50	0.000078786573	438089.3333	1.666666667	438693.6667	0.3333333333	0.9999977189	0.9999961934	0.9999992387	0.9999992387	0.000003796905067	0.9999977209	0.999997716
Full, 10/90**	0	788988.6667	0	4145819.333	0	1	1	1	1	0	1	1
Full, 50/50	0	438245	0	2303315	0	1	1	1	1	0	1	1
Support Vector Machine												
Balanced, 10/90	0.3454495392	785764.6667	12357.66667	776663.6667	3427	0.9899983927	0.9845167336	0.995657616	0.995657616	0.01566200899	0.9899978035	0.9900557634
Balanced, 1/99	1.084273022	851271	37727	830265	16771	0.9686075838	0.9575636143	0.9806791282	0.9575623342	0.04243766577	0.9686072456	0.9689830033
Full, 10/90	0.6750616614	73370	4509	410466	5136	0.9804551746	0.9421024923	0.9345782488	0.9421024923	0.05789750767	0.961856267	0.938325287
Full, 1/99	0.6864640375	810307	50294	4510095	57593	0.9801250449	0.9415594451	0.9336409725	0.9555123969	0.04395696398	0.9613062637	0.93758349
Multi-Layer Perceptron												
Balanced, 10/90	0.2389558039	785339.3333	7215.666667	781955	3703	0.9930816267	0.9909116133	0.995306963	0.995306963	0.009143482251	0.9930817404	0.9930979742
Balanced, 50/50	0.2402192313	436545.3333	3952.333333	434141.6667	2145.666667	0.9930450453	0.9910280143	0.9951094762	0.9951094762	0.009021524284	0.993043976	0.9930638862
Full, 10/90	0.1824718813	781671.3333	18475.66667	4127066	7595	0.9947169846	0.9769519708	0.9903769239	0.9903769239	0.004456763691	0.9929600801	0.9835983862
Full, 50/50	0.1547505492	434253.6667	8269.666667	2295023	4013.666667	0.9955195825	0.9813775558	0.9908429423	0.9908429423	0.003590542637	0.9936261998	0.9860697416

Graph Invariant:	**Is Regular** (or k-regular)											
Data Structure:	Directed graph with 4 nodes											
Valid if:	Each node in the graph has k ingoing and k outgoing edges											
Model					**Metrics**							
Valid/Invalid, Train/Test	BCEL	True Positive	False Positive	True Negative	False Negative	Accuracy	Precision	Recall	True Positive Rate	False Positive Rate	Area Under Curve	F1 Score
Decision Tree												
Balanced, 10/90	0.257829119	89484.66667	799	89171.66667	544.6666667	0.9925351852	0.9911504288	0.993950355	0.993950355	0.008880784057	0.9925347855	0.9925483498
Balanced, 50/50	0.1763797852	49740.66667	221	49748.66667	289.6666667	0.9948933333	0.9955765061	0.9942101469	0.9942101469	0.004422523599	0.9948938116	0.9948928451
Full, 10/90	0.4549704038	13759.66667	104656.6667	13536180.33	76213.33333	0.9868274341	0.1162025886	0.1529320991	0.1529320991	0.007672305311	0.5726298969	0.1320606455
Full, 50/50	0.4456110332	5072.666667	53564.33333	7524739	44852	0.9870984017	0.08650543841	0.1016062105	0.1016062105	0.007068117278	0.5472690466	0.09344914186
Support Vector Machine												
Balanced, 10/90 **	0.128241239	89370	9.333333333	89961.66667	659	0.996287037	0.9998955764	0.9926800934	0.9926800934	0.0001037377303	0.9962881778	0.9962747679
Balanced, 50/50	0.1238791766	49598.66667	12.33333333	50042.66667	346.3333333	0.9964133333	0.9997513439	0.9930657255	0.9930657255	0.0002463309294	0.9964096973	0.9963973149
Full, 1/99 (only 1 iteration)	0.226367473	0	0	15080682	99491	0.9934459904	0	0	0	0	0.5	0
Full, 10/90												
Multi-Layer Perceptron												
Balanced, 10/90	0.1425051156	89456	153.6666667	89801.33333	589	0.9958740741	0.9982851823	0.9934587598	0.9934587598	0.001708126773	0.9958753165	0.9958661012
Balanced, 50/50	0.1008538134	49674	73.33333333	50034	218.6666667	0.99708	0.9985257253	0.9956169263	0.9956169263	0.001463188242	0.997076869	0.997069102
Full, 10/90	0.2420176263	2874.333333	9051.333333	13631722.33	87162	0.9929928873	0.1472819091	0.03194259922	0.03194259922	0.0006635472233	0.515639526	0.04917174535
Full, 50/50	0.2369364125	495.3333333	2901.333333	7575403	49428.33333	0.993139997	0.2258733896	0.009917297864	0.009917297864	0.0003828483836	0.5047672247	0.01771452391

Graph Invariant:	**Is Dense**											
Data Structure:	Directed graph with 7 nodes											
Valid if:	The density of the graph (E/V^2) is greater than .70 (70% of max density)											
Model					**Metrics**							
Valid/Invalid, Train/Test	BCEL	True Positive	False Positive	True Negative	False Negative	Accuracy	Precision	Recall	True Positive Rate	False Positive Rate	Area Under Curve	F1 Score
Decision Tree												
Balanced, 10/90	2.109775077	88413.66667	6091	87805	5373.333333	0.9389165064	0.9355500094	0.9427048371	0.9427048371	0.06487007446	0.9389173813	0.9391113831
Balanced, 50/50	1.578535255	49754.66667	2415.666667	49748	2349.666667	0.9542972596	0.9536971871	0.9549050593	0.9549050593	0.04630937712	0.9542978411	0.9542995387
Full, 10/90	0.7168588743	73905	19321.66667	1774358.333	19852	0.9792450468	0.7927523475	0.7882633016	0.7882633016	0.0107720802	0.8887456107	0.7904939623
Full, 50/50	0.6552799931	42214.66667	9966.666667	986467.6667	9927	0.981027921	0.8089998823	0.8096149942	0.8096149942	0.01000233043	0.8998063319	0.8093069694
Support Vector Machine												
Balanced, 10/90	1.063044679	77395	3326	75588	1532	0.969222192	0.9587963479	0.9805896588	0.9587968584	0.04120314164	0.9692212556	0.9695705552
Balanced, 1/99	1.061320083	851287	36639	831402	16705	0.9692721249	0.958736396	0.9807540639	0.9587364262	0.04126357376	0.969272445	0.9696200325
Full, 10/90	0.6992784749	73529	4660	409961	5331	0.9797540331	0.9404008236	0.9323991884	0.9404008236	0.05959917635	0.9605800043	0.9363829123
Full, 1/99	1.429588527	69983	6106	450378	16362	0.9586094332	0.9197518695	0.810504372	0.9197518695	0.08024813047	0.8985641093	0.8616792051
Multi-Layer Perceptron												
Balanced, 10/90**	9.99E-16	93759.33333	0	93923.66667	0	1	1	1	1	0	1	1
Balanced, 50/50	9.99E-16	52092.33333	0	52175.66667	0	1	1	1	1	0	1	1
Full, 10/90	9.99E-16	93844.33333	0	1793592.667	0	1	1	1	1	0	1	1
Full, 50/50	9.99E-16	52166.66667	0	996409.3333	0	1	1	1	1	0	1	1