TESTING ARTIFICIAL INTELLIGENCE-BASED SOFTWARE SYSTEMS

by

JAGANMOHAN CHANDRASEKARAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2021

To Amma, Appa and Ankita

ACKNOWLEDGEMENTS

I consider myself fortunate enough to have an opportunity to pursue doctoral studies. First and foremost, I would like to acknowledge my deepest gratitude and appreciation to my advisor, Professor Dr. Jeff Lei. This dissertation would not have happened without his patience, support, and guidance. I was a novice when I joined his group; Still, he was kind enough to patiently help me navigate graduate school while not compromising on the quality of work. I am deeply grateful for his valuable guidance and for instilling the value of hard work in me. I learned significantly through our interactions, and his feedback over the years pushed me to become a better researcher. I would like to extend my gratitude to my committee members: Dr. Christoph Csallner, Dr. David Kung, and Professor. David Levine for their interest in my research and for providing valuable feedback.

I would like to thank all my current and past colleagues from the Software Engineering Lab: Dr. Laleh Sh. Ghandehari, Dr. Jing Xu, Dr. Feng Duan, Dr. Huadong Jack Feng, Michael Hao, Dr. Sarker Tanve Ahmed Rumee, Chunxia Sui, Xiaolei (Harry) Ren, Mengfei (Angela) Ren, Sunny Shree, Qiping Wei, Edrik Aguilera, and Ana Jovanovic for all their help and valuable feedback over the years.

I would like to extend special thanks to our collaborators from the National Institute of Standards and Technology (NIST) - Dr. Richard Kuhn and Dr. Raghu Kacker. They had always been welcoming and supportive. Thanks to all the anonymous reviewers whose feedback helped to refine and improve my work.

I want to thank Dr. Bahram Khalili for all his support and advice throughout my graduate school. Thanks to him for believing in me and giving me an opportunity; I couldn't have pursued the doctoral studies without his timely guidance. Special thanks to Dr. John Robb, with whom I

had the opportunity to work as his GTA for around four years. Professor. Robb was very supportive and accommodating, that allowed me to balance my research work and GTA responsibilities with ease. Many thanks to the CSE department staff, especially Ms. Pam Mcbride, Ms. Sherri Gotcher, and Ms. Ginger Dickens, for helping me with the administrative process and making my stay easier at UTA.

I would like to extend my gratitude to music streaming services – Apple Music and Spotify for making music available 24x7. Especially the music of Ludovico Einaudi and Above & Beyond kept me refreshed and helped me to focus, especially during deadlines. Staying away from the family can be isolating and challenging. My journey as a graduate student was certainly made lighter by the brilliant late-night comedians and their team. Thanks to Stephen Colbert, Seth Meyers, John Oliver, Jimmy Kimmel, and Jimmy Fallon for keeping me sane and making me laugh every other night. Through their shows, I learned the art of explaining complex topics in a simple, understandable form.

Thanks to my dear friends at UTA: Danny John, Poornima Gopala, and Dr.Abhishek Chatterjee, who helped and motivated me to apply to the Ph.D. program. Specifically, I would like to thank Abhishek, my apartment-mate, for six years. I will always cherish our conversations that spanned across a wide range of topics from politics to philosophy. Our discussions certainly helped me to broaden my thought process. Thanks to him for taking an interest in my research and being open to discuss and provide critical feedback on my research ideas, reading and reviewing my manuscripts, and helping me become a better writer.

I would also like to thank everyone I had the opportunity to meet and interact with over the past six years at UTA. Special thanks to Akilesh for helping me set up GPUs for my experiments,

Ashwin Raju for always being open and helpful in solving my doubts, and Ragul for providing critical help at the time of need.

Thanks to my uncle, Göran Wiking, for gifting me a Toshiba laptop, my first ever computer. That laptop planted the seed of curiosity about hardware and software in me. I would like to express my deepest gratitude to my dear friends: Vinoth, Amarnath, Someshwar, and R.R. Dhinesh, who was always there for me and motivated me to pursue greater goals. I would also like to thank all my friends from various phases of life who helped me either directly or indirectly in this journey: Karthik, Raja, Saravana kumar, Arun, Pratik, Dinesh, Pandiarajan, Krishnakumar, Ashima Batra, Archana Ashok, Shelly Chaudhary, Lalitha Akilanathan and Thohai Balaiya. I would like to thank my late grandparents, uncles, and aunts who helped me along in this journey.

I am hugely indebted to my Amma, Mrs. Malathi Chandrasekaran, and my Appa, Mr. R.M. Chandrasekaran, for their sacrifices. My education was their ultimate priority; They consistently went above and beyond to ensure that I have the best possible opportunities in my life. Their constant support and motivation helped me to reach where I am today. Thanks to my Amma for the constant nudge and for ensuring that I submit my manuscripts on time. Thanks to my Appa for instilling the values of seeing the bigger picture in life. Safe to say, without their conviction, support and motivation, it would have been impossible for me to pursue doctoral studies.

Last but certainly not least, thanks to my wife, Ankita. Thank you for being patient and supportive when I was busy with submission deadlines, offering to read and edit my manuscript, and helping in any way she could. I will be forever indebted for her thoughtfulness, motivation, and countless sacrifices that enabled me to complete this journey successfully.

July 15, 2021

ABSTRACT

TESTING ARTIFICIAL INTELLIGENCE-BASED SOFTWARE SYSTEMS

Jaganmohan Chandrasekaran, PhD

The University of Texas at Arlington, 2021

Supervising Professor: Jeff (Yu) Lei

Artificial Intelligence (AI)-based software systems are increasingly used in high-stake and safety-critical domains, including recidivism prediction, medical diagnosis, and autonomous driving. There is an urgent need to ensure the reliability and correctness of AI-based systems. At the core of AI-based software systems is a machine learning (ML) model that is used to perform tasks such as classification and prediction.

Unlike software programs, where a developer explicitly writes the decision logic, ML models learn the decision logic from a large training dataset. Furthermore, many ML models encode the decision logic in the form of mathematic functions that can be quite abstract and complex. Thus, existing software testing techniques cannot be directly applied to test AI-based applications.

The goal of this dissertation is to develop methodologies for testing AI-based software systems. This dissertation makes contributions in the following areas: **Test input generation:** 1) A combinatorial approach for generating test configurations to test five classical machine learning algorithms. 2) A combinatorial approach for generating test data (synthetic images) to test Deep Neural Network (DNN) models used in autonomous driving cars. **Test Cost Reduction:** 3) An empirical study that analyzes the effect of using sampled datasets to test supervised learning algorithms. **Explainable AI (XAI):** 4) A software fault localization-based explainable AI (XAI)

approach that produces counterfactual explanations for decisions made by image classifier models (DNN models).

This dissertation is presented in an article-based format and includes five research papers. The first paper reports our work on applying combinatorial testing to test five classical machine learning algorithms. The second paper reports an extensive empirical evaluation of testing ML algorithms with sampled datasets. The third paper introduces a combinatorial testing-based approach to generating test images to test pre-trained DNN models used in autonomous driving cars. The fourth paper is an extension of the third paper. This paper presents an initial study that evaluates the performance of combinatorial testing in testing DNNs used in autonomous driving cars. The fifth paper presents an explainable AI (XAI) approach that adopts BEN, an existing software fault localization technique, and produces explanations for decisions made by ML models. All five papers have been accepted at peer-reviewed venues. Paper 1, Paper 2, Paper 3, and Paper 5 have been published, while Paper 4 is currently in press.

Table of Contents

List of Illustrations

List of Tables

CHAPTER 1. INTRODUCTION

Artificial Intelligence (AI) based software systems are increasingly used across application domains. The principal component within all AI-based software systems are the machine learning (ML) models that perform various prediction and classification tasks. To build an ML model, the practitioner typically uses a machine learning framework such as WEKA [2], TensorFlow [1], Keras [3], and Pytorch [4]. A machine learning framework implements a collection of machine learning algorithms. The practitioner chooses a machine learning algorithm from a framework and provides two types of input: a dataset and hyperparameters (a set of configurable parameters used to fine-tune the model's learning process) to the algorithm. The algorithm then examines the data to discover insights from hidden patterns (algorithm learns from the dataset) and derives a decision logic referred to as a machine learning model. The process of creating an ML model is referred to as training or building a model. An AI-based software system consists of one or more ML models.

1.1. RESEARCH OVERVIEW

Traditional software applications have their computational logic written explicitly by humans. Compared to this, ML models derive their decision logic based on a dataset. As presented in Figure 1-1, a combination of the dataset, hyperparameters and model's architecture (ML algorithm) impact the behavior of an ML model. Furthermore, many ML models encode the decision logic in the form of mathematic functions that can be quite abstract and complex, and their correctness is determined using a statistical score. Thus, existing software testing techniques cannot be directly applied to test AI-based software systems. With broader adoption of AI-based software systems across domains, including safety- critical systems such as medical imaging,

autonomous driving, and aviation, there is a need to develop techniques, approaches, and tools to systematically test AI-based software systems.

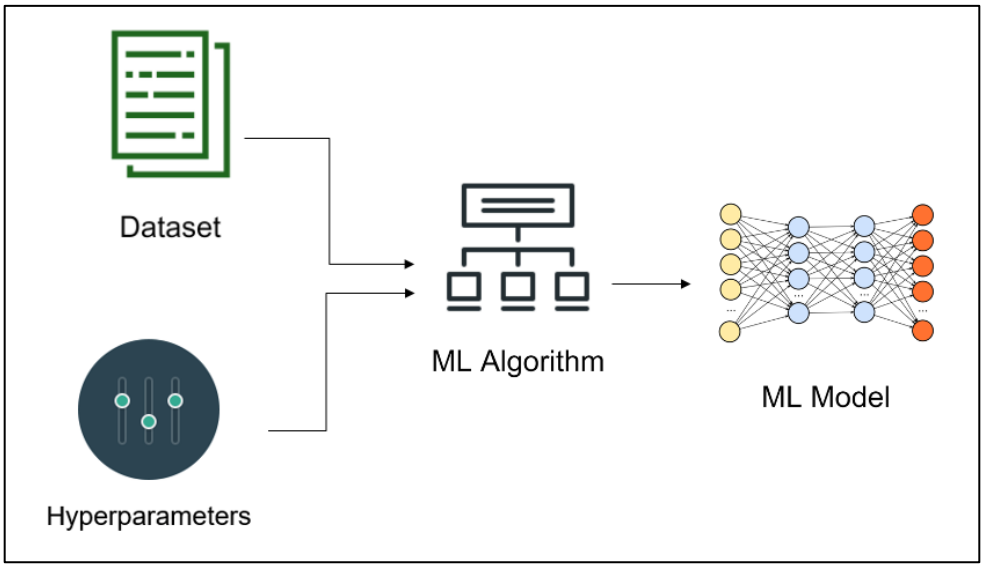


FIGURE 1-1 ML MODEL DEVELOPMENT WORKFLOW

Figure 1-1 presents the ML model development workflow, and they can be broadly classified into two groups: 1). Software components (dataset, hyperparameters, and ML algorithm) used to create an ML model, and 2). a trained ML model. From a testing perspective, it is imperative to test each component (from the ML model development workflow) to guarantee the quality of AI-based software systems. In general, a fault can arise from

- Hyperparameters used to train the ML model,
- Training dataset (incorrect data, over/under-representation of class labels),
- Fault in the ML training algorithm, and
- Fault in the model.

This dissertation aims to deconstruct various aspects of ML model development workflow into individual components and present solutions to improve the testing procedure for each component.

The first part of the dissertation (Chapter 2 and Chapter 3) is devoted to testing the software components (hyperparameters, dataset, ML algorithm) that produce an ML model. In Chapter 2, we present an approach that generates test cases based on the hyperparameters to test five classical ML algorithms. To speed up the testing process, in Chapter 3, we investigate the effect of sampled datasets in testing supervised ML algorithms. The remainder of the dissertation focus on testing the pre-trained ML models. The third project (Chapter 4) focuses on generating test inputs to test pre-trained Deep Neural Network (DNN) models used in autonomous cars. The fourth project (Chapter 5), an extension of the previous project (Chapter 4), reports a preliminary study that evaluates the performance of combinatorial testing in testing DNNs used in autonomous cars. Finally, the fifth project (Chapter 6) presents an approach to produce explanations for decisions made by ML models.

1.2. SUMMARY OF PUBLICATIONS

This dissertation is presented in an article-based format and includes five research papers. In Chapter 2, we present the paper titled, *Applying Combinatorial Testing to Data Mining Algorithms*, which was published in IEEE 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2016. This paper is co-authored and includes the following authors: Huadong Feng, Yu Lei, Raghu Kacker, and D. Richard Kuhn. As the primary author, I was responsible for designing and conducting the experiments, analyzing the results for three ML algorithms while Dr. Feng was responsible for the other two ML algorithms. Dr. Feng and I equally contributed to the manuscript. Dr. Lei supervised the project by providing

feedback on the research directions, reviewing and editing the manuscripts. Dr. Kacker and Dr. Kuhn provided feedback in revising and improving the final version of the paper.

The hyperparameters govern how an ML algorithm learns from a given dataset. These hyperparameter values determine which part of a given dataset is reflected more or less in the final trained model. Therefore, if there exist certain faults in a trained model due to a lack of representation of certain portions of a given data set, then it is imperative that these faults could be attributed to the hyperparameter values. This paper presents an approach that applies combinatorial testing and generates test inputs based on the hyperparameters (referred to as configuration options in WEKA) of ML algorithms implemented in WEKA, an open-source machine learning workbench. The result from this study suggests combinatorial testing is effective in testing the hyperparameters of ML algorithms.

Chapter 3 presents a paper titled, *Effectiveness of dataset reduction in testing machine learning algorithms*. The paper was published in IEEE 2nd International Conference on Artificial Intelligence Testing (AITest), in 2020. This paper is co-authored and includes the following authors: Huadong Feng, Yu Lei, Raghu Kacker, and D. Richard Kuhn. I am the primary author of this paper. As the lead on this project, I was responsible for designing and conducting the experiments, collecting and analyzing the coverage results. Dr. Feng helped in writing test scripts to automate the mutation testing experiments. Dr. Lei supervised the project by formulating the research questions and helped significantly to improve the manuscript. Dr. Kacker and Dr. Kuhn provided feedback in revising and improving the final version of the paper.

Unlike traditional software applications, testing ML algorithms can be very expensive and time-consuming as they typically take a longer time to execute. It would be very useful in practice if the execution time of ML algorithms could be reduced for testing purposes. One of the possible

reasons for a longer execution time of ML algorithms can be attributed to the high volume (large) input dataset. Therefore, in this paper, we explored the use of sampled datasets via different sampling techniques in testing machine learning algorithms. Our experimental results suggest that sampled datasets can accelerate the testing phase of ML applications while largely preserving the fault detection effectiveness of the original datasets.

Chapter 4 presents a paper titled, *A Combinatorial Approach to Testing Deep Neural Network-based Autonomous Driving Systems.* The paper was published in IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2021. This paper is co-authored and includes the following authors: Yu Lei, Raghu Kacker, and D. Richard Kuhn. I am the primary author of this paper, and I was responsible for developing the test generation approach, designing input parameter models, generating abstract t-way test sets, writing automation scripts to generate synthetic images, conducting experiments, and analyzing the results. Dr. Lei supervised the project and helped in designing the evaluation metrics, reviewing, and editing the manuscript. Dr. Kacker and Dr. Kuhn contributed to improving the final draft of the paper.

Autonomous systems such as self-driving cars use pre-trained ML models to perform intelligent tasks like pedestrian detection, steering control, and lane control. Despite its promising potential, ML models fail to exhibit expected behavior in real-world scenarios and resulting in fatalities in some cases. There is a need to test ML models with realistic driving scenarios before deploying them in the real world. This paper presents a combinatorial approach to generate synthetic images (test data) to test Deep Neural Network (DNN) models used in self-driving cars. The results from this work suggest that the combinatorial testing approach can be very effective for testing autonomous driving systems. The proposed approach was able to successfully detect a

significant number of inconsistent behaviors in pre-trained DNN models developed to predict the steering angle of a car.

Chapter 5 presents a paper titled, *Evaluation of T-Way Testing of DNNs in Autonomous Driving Systems.* This paper is accepted at the IEEE 3rd International Conference on Artificial Intelligence Testing (AITest) in 2021. This paper is co-authored and includes the following authors: Ankita Ramjibhai Patel, Yu Lei, Raghu Kacker, and D. Richard Kuhn. As the primary author of this paper, I was responsible for designing and conducting the comparison study. Ms. Ankita Ramjibhai Patel helped in sorting and analyzing the neuron coverage results. Dr. Lei helped in designing the comparison study, and in reviewing and editing the manuscript.

In this paper, we present a preliminary study that evaluates the performance of a combinatorial testing-based approach in testing DNNs used in autonomous cars. In this study, we compare the synthetic images generated using the combinatorial approach to DeepTest [5], a state-of-the-art test generation tool that aims at generating synthetic images that maximize neuron coverage, a measure of the proportion of neurons activated in a DNN model. The results from this study suggests that the combinatorial approach generates valid synthetic images and can cover more neurons than the synthetic images generated by the DeepTest approach.

Chapter 6 presents a paper titled, *A Combinatorial Approach to Explaining Image Classifiers* which was published in IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2021. This paper is co-authored and includes the following authors: Yu Lei, Raghu Kacker, and D. Richard Kuhn. I am the primary author of this paper. I was responsible for developing the approach, designing, and conducting the experiments, and analyzing the results. Dr. Lei supervised the project, and his critical feedback

helped significantly strengthened the overall project. Dr. Kacker and Dr. Kuhn reviewed the manuscript, and their feedback helped us refine the final version of the paper.

This paper presents an explainable AI (XAI) approach to produce counterfactual explanations for decisions made by image classifier models. ML models, which are black boxes by nature, do not provide any explanation behind their decision. The lack of explanation limits the oversight on these models and prevents human users from diagnosing and repairing biases or undesirable behaviors. This considerably reduces the trustworthiness of AI-based software systems and restricts their deployment towards applications in the real world. Hence, in the AI research community, a significant amount of effort is being invested on the development of methods for deriving explanations behind decisions made by AI systems. We observe that generating an explanation for a machine learning model is similar to fault localization, a classical problem in software engineering. We propose an XAI approach that adopts BEN, a software fault localization-based tool for deriving counterfactual explanations for image classifier models. The results suggest the proposed approach can successfully generate counterfactual explanations for DNN based image classifiers.

Chapter 7 presents the concluding remarks and directions for our future work.

1.3. REFERENCES

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16) (pp. 265-283).
2. Eibe Frank, Mark A. Hall, and Ian H. Witten (2016). The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.

3. Keras-team/Keras: Deep Learning for humans: https://github.com/keras-team/keras, Accessed: 2020-11-18
4. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Desmaison, A. (2019). Pytorch: An imperative style, high-performance deep learning library. In Advances in neural information processing systems (pp. 8026-8037).
5. Tian, Y., Pei, K., Jana, S., & Ray, B. (2018, May). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In Proceedings of the 40th international conference on software engineering (pp. 303-314).

Chapter 2. Applying Combinatorial Testing to Data Mining Algorithms

The chapter contains a paper published in the IEEE 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2017.

Applying Combinatorial Testing to Data Mining Algorithms[*]

Jaganmohan Chandrasekaran[1], Huadong Feng[1], Yu Lei[1], D. Richard Kuhn[2], Raghu Kacker[2]

[1]Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, USA

Abstract — Data mining algorithms are used to analyze and discover useful information from data. This paper presents an experiment that applies Combinatorial Testing (CT) to five data mining algorithms implemented in an open-source data mining software called WEKA. For each algorithm, we first run the algorithm with 51 datasets to study the impact different datasets have on the test coverage. We select one dataset that achieves the highest branch coverage. Next we construct positive and negative combinatorial test sets of configuration options and execute each test set with the selected dataset. Test effectiveness is measured using branch and mutation coverage. Our results suggest that when testing data mining algorithms: (1) larger datasets do not necessarily achieve higher coverage than smaller datasets; (2) test coverage increases progressively slower as test strength increases; and (3) branch coverage correlates well with mutation coverage.

Keywords—Combinatorial Testing; Data mining; Machine learning; Input parameter modeling; Branch coverage; Mutation testing;

2.1. INTRODUCTION

Big data applications are becoming more popular as large amounts of data are generated and collected in virtually every domain, e.g., e-commerce, social networking, and scientific computing [11,12]. These applications typically employ data mining algorithms to analyze data and discover useful information. Data mining algorithms include supervised learning algorithms, and un-supervised learning algorithms to perform tasks such as classification, clustering, association rule mining [36].

In this paper, we present an experiment that applies combinatorial testing (or CT) to software that implements data mining algorithms. Testing data mining algorithms has several challenges. First, data mining software typically involves complex computation and decision logic. This is because data mining algorithms can be quite sophisticated. Second, data mining software often deals with datasets that have complex structure. Thus, it can be difficult to model and characterize the input space. Third, many data mining algorithms are designed to process large amounts of data. However, it is impractical to test large amounts of data at the development stage when testing is frequently performed. In the remainder of this paper, we will refer to data mining algorithms and software that implements data mining algorithms simply as data mining algorithms, unless otherwise specified.

The goal of our experiment is to evaluate the effectiveness of CT for testing data mining algorithms. In our experiment, we apply CT to five data mining algorithms implemented in the Waikato Environment for Knowledge Analysis (WEKA) tool. These five algorithms include C4.5, K-Means, SVM, Apriori and EM, and they are identified to be the top five most influential data mining algorithms by the IEEE International Conference on Data Mining (ICDM) [44]. Each algorithm takes two types of input, including a dataset to be analyzed, and configuration options

that are used to customize the behavior of the algorithm. In our experiment, the input dataset for each algorithm was selected from a collection of 51 public benchmark datasets provided by WEKA and UC Irvine [32].

To carry out our experiment, we first ran each algorithm with the default configuration on the 51 datasets to study the impact different datasets have on the test coverage, and selected one dataset that achieved the highest branch coverage. We then created an input parameter model (IPM) for the configuration options of each algorithm. An IPM consists of representative values of each configuration option as well as constraints that exist between these values. We performed both positive and negative testing of each algorithm using the selected dataset and the IPM of the configuration options. For positive testing, we created 1-way to 6-way test sets using the valid values in the IPM. For negative testing, we created a 1- way test set for the invalid values in the IPM. In the 1-way negative test set, each invalid value is covered by one test in which every other value is a valid value. Test effectiveness is measured in terms of both branch coverage and mutation coverage.

The major results of our experiment are summarized as follows:

• Larger datasets do not necessarily achieve higher test coverage than smaller datasets. The sizes of the datasets that are applicable to each algorithm range from as few as 14 instances to as many as 20,000 instances. However, almost all of these datasets achieved similar branch coverage. In some cases, very small datasets achieved higher coverage than very large datasets. For example, for algorithm Apriori, the *weather .nominal* dataset has only 14 instances, but it achieved higher coverage than the mushroom dataset, which has 8124 instances. This suggests that the size of a dataset is not a dominating factor in deciding test coverage. Other factors, e.g., structure of a dataset, and relationship between different instances, might play a more significant role.

• Test coverage of CT test set increases progressively slower with respect to increase of test strength. In our experiment, test coverage increases more significantly when test strength increases from 1-way to 3-way. After 4-way testing, higher strength test sets no longer provide significant coverage improvement. This result is consistent with the results of other empirical studies that apply CT to general software applications.

• Branch coverage correlates well with mutation coverage. The results of our experiment suggest that in general, branch coverage correlates with mutation coverage. In particular, higher branch coverage often implies higher mutation coverage. This suggests that branch coverage could be used as a good indicator of fault detection effectiveness for data mining algorithms, since mutation coverage is expensive to measure.

To the best of our knowledge, our work is the first attempt to evaluate the effectiveness of CT to data mining algorithms. In general, little work has been reported on testing data mining algorithms. We believe that our experiment provides initial insights that can be useful for developing more effective testing techniques for data mining algorithms.

The remainder of this paper is organized as follows. Section 2.2 discusses the major design decisions made in our experiment. Section 2.3 presents the major results obtained from our experiment. Section 2.4 briefly reviews related work, including existing work on CT and on testing data mining algorithms. Section 2.5 provides our conclusion and outlines several directions for future work.

2.2. EXPERIMENTAL DESIGN

In this section, we present the design of our experiments. We formulate our research questions, identify the subject algorithms and datasets, present our approach to Input Parameter Modeling (IPM) and test generation, and discuss the metrics used to measure test effectiveness.

2.2.1 RESEARCH QUESTION

The goal of this project is to evaluate effectiveness of CT applied to data mining algorithms. We formulate the following research questions:

1) How do different datasets impact test coverage?

2) How effective is CT applied to data mining algorithms?

3) Is branch coverage a good indicator of fault detection effectiveness?

2.2.2 SUBJECT PROGRAMS

WEKA is one of the most widely used data mining tools. WEKA is developed by University of Waikato, and implements a collection of data mining algorithms as different packages. The subject programs include five data mining algorithms implemented in the WEKA tool: (1) C4.5, which is a supervised learning algorithm that takes a collection of cases as input, and output a classifier that predicts the class to which a new case belongs using decision tree[44]; (2) K-Means, which is an unsupervised learning algorithm that performs clustering by partitioning a given dataset into *k* clusters such that the members of each cluster are similar to each other; (3) SVM, which is a supervised learning algorithm that uses the vector space to build a SVM classification model. The model predicts the class to which a new case belongs; (4) Apriori, which is an unsupervised learning algorithm that generates association rules by identifying frequent item sets; and (5) EM, which is an unsupervised learning algorithm that uses statistical models to perform clustering. These five algorithms are identified to be the top five most influential data mining algorithms [44].

Table 2-1 shows information about the WEKA packages that implement the five algorithms.

TABLE 2-1 WEKA PACKAGE INFORMATION

Algorithm Name	Package Name in WEKA	# of Files	# of Classes	# of Branches	LOC	# of Configuration Parameters	# of Applicable Datasets
Apriori	weka.associations	5	5	580	1349	12	11
EM	weka.clusterers	6	10	736	1825	14	46
C4.5	weka.classifiers.trees.J48	17	17	696	1641	17	44
K-Means	weka.clusterers	5	7	699	1721	18	46
SVM	libsvm	6	18	1124	2138	17	44

2.2.3 DATASETS

We selected our datasets from a collection of 51 public benchmarking datasets provided by WEKA and UC Irvine [32]. Table 2-2 shows the statistics of the 51 subject datasets.

Different algorithms require different types or formats of data. As a result, not every dataset is applicable to every algorithm. To determine the applicability of a dataset to a given algorithm, we run the dataset with the algorithm. A dataset is considered applicable to an algorithm if executing the dataset with the algorithm provides meaningful output without any exception. The number of applicable datasets for each algorithm is shown in the last column of Table 2-1.

TABLE 2-2 DATASET INFORMATION

	# of Attributes	**# of Instances**	**Size in KB**
Maximum	217	20000	1978.39
Minimum	2	14	0.483398
Average	23.92157	1466.902	229.5591
Standard Deviation	31.73506	3079.593	393.0123
Median	18	604	44.82715

2.2.4 INPUT PARAMETER MODELING

Before CT is applied, we must create the input parameter model (IPM) [21]. Each subject algorithm takes two types of input, including a dataset to be analyzed, and a set of configuration options that are used to customize the behavior of the algorithm. Our experiment focuses on CT of configuration options. As mentioned in Section V, CT of datasets is left for future work. Thus our modeling process mainly consists of identifying representative values for different configuration options.

In the following, we use the *Apriori* algorithm as an example to explain our approach. We categorize configuration options into two groups.

• *Group 1*: This group includes options with a set of predefined choices. For each option in this group, every predefined choice is identified as a representative value for this option.

Figure 2-1 shows some configuration options of the Apriori algorithm. Consider as an example option "- T", which is used to specify the metric type. This option has four predefined choices, *Confidence, Lift, Conviction, and Leverage,* each of which is identified to be a representative value for this option.

-N <required number of rules output>

The required number of rules. (default = 10)

-T <0=confidence | 1=lift | 2=leverage | 3=Conviction>

The metric type by which to rank rules. (default = confidence)

-C <minimum metric score of a rule>

The minimum confidence of a rule. (default = 0.9)

-c <the class index>

The class index. (default = last)

FIGURE 2-1 CONFIGURATION OPTIONS FOR APRIORI

Group 2: This group includes options that do not have a set of predefined choices. Instead, the user can input any value that is valid. For each option in this group, equivalence partitioning is used to identify representative values.

We observe that in our subject programs, all the options in this group are of type Integer, Float, Double. We first identify boundary values that distinguish valid and invalid values, as shown in Figure 2-2. A boundary value itself may or may not be valid. In Figure 2-2, a square bracket indicates a valid boundary value, and a parenthesis indicates an invalid boundary value. Next, we partition valid and invalid values into different groups as needed, based on domain knowledge.

The set of representative values include one representative value from every partition of valid and invalid values.

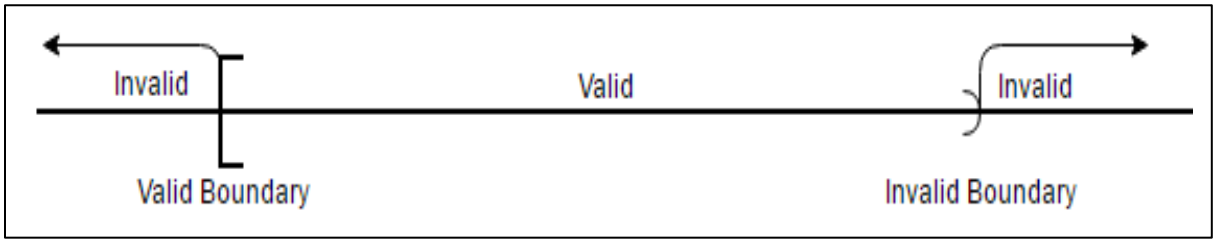


FIGURE 2-2 EQUIVALENCE PARTITIONING FOR GROUP 2 CONFIGURATION OPTIONS

Consider as an example the - *C* (minimum metric score) option in Figure 2-1. This option allows the user to specify a threshold value for the selected metric type. Assume that the user chooses Confidence as the metric type, i.e., *-T 0 (Figure 2-1)*. Based on domain knowledge, we identify the boundary values for minimum metric score as *0* and *1*.

Next we identify the equivalence classes for valid and invalid values and select a representative value from each class. For valid values, we identify three equivalence classes: (1) {values that take every possible rule}, (2) {values that only take rules with 100% confidence}, and (3) {other values, i.e. values that do not take every possible rule, and do not require 100% confidence for each rule}. The first class consists of a single value, i.e., 0. Similarly, the second class consists of a single value, i.e., 1. The third class includes every value that is greater than 0 and less than 1. Thus, we select the following three representative values, including 0, 1, and 0.9. Note that "0.9" is the default value of this option as shown in Figure 2-1. In general, the default value is selected as the representative value for the equivalence class that contains the default value. Doing so helps to reduce number of representative values identified for each parameter.

For invalid values that are outside of the boundary values, we identify two equivalence classes: (1) {*value* | *value* < *lower boundary*}; and (2) {value | *value* > *higher boundary*}. A random value can be chosen from each of the two equivalence classes as the representative values.

In addition to identifying representative values for each configuration option, we have also identified constraints between different values. Constraints are used to prevent ACTS [46] from generating invalid combinations. For example, in the Apriori algorithm, when option "-A" is true, the only allowed metric type "-T" is *confidence*, i.e., option "- T" must take the value of 0. Table 2-3 shows the constraints that are identified for algorithm Apriori.

TABLE 2-3 CONSTRAINTS IDENTIFIED FOR APRIORI

A = false => c = -1
A = true => T = 0
T = 0 => (T = 0.1 || C = 0.9)
T = 1 && A = false => (M = 0.1 || M = 0.95)
T = 1 || T = 3 => (T =1.1 || T = 1.5)
T = 2 => (C = 0.1 || C = 0.5)

2.2.5 TEST GENERATION

We performed both positive and negative testing in our experiments. For positive testing, we created test sets that achieve 1-way to 6-way coverage for valid values using the *extend* mode from ACTS. The *extend* mode allows a test set to be built by extending an existing test set. By using the *extend* mode, every higher strength test set will be the superset of its lower strength test set(s). For negative testing, we generated a test set that achieves 1-way coverage for invalid values. That is, each invalid value is covered by one and only one test in which every other value is a valid

value. Note that in order to avoid potential mask effects, a negative test should contain at most one invalid value.

TABLE 2-4 SIZES OF TEST SETS

	Apriori	**EM**	**J48**	**SimpleKMeans**	**LibSVM**
1-way	7	3	4	4	5
2-way	33	11	14	16	21
3-way	132	37	48	49	76
4-way	478	91	133	136	232
5-way	1440	214	349	368	637
6-way	4055	463	835	911	1546
Negative	12	18	9	11	15

We used ACTS to generate both positive and negative test sets. Table 2-4 shows the sizes of test sets of different strengths. Since the representative values are abstract values, the tests generated by ACTS are abstract tests. These abstract tests need to be translated to concrete tests prior to execution. For example, consider the option, "-c", representing Class Index, as shown in Figure 1. String "last" is an abstract value of this option that represents the last column (or attribute) of the input dataset. This abstract value must be mapped to the actual index of the last column in a dataset.

For each algorithm, we have written a script that performs automatic translation from abstract tests to concrete tests. The corresponding concrete value of an abstract value is calculated based on the selected input dataset. For example, abstract value "last" for option, -c, the concrete value when executing *weather.nominal* dataset for Apriori will be set to the actual last index, "5".

2.2.6 METRICS

In our experiments, we used branch coverage and mutation coverage to measure test effectiveness. We used JaCoCo to record branch coverage. JaCoCo is a free Eclipse plugin that measures statement and branch coverage at the byte code level [22].

We used an open-source mutation testing tool called PIT to measure mutation coverage [14]. We selected all available mutators that are provided by PIT for generating mutants. PIT uses JUnit tests to determine whether a mutant is killed. All JUnit tests must pass before PIT can be applied. We first ran each test case with the original programs and stored the output as the expected output. Then we created JUnit tests that check the actual output against the expected output. Whenever a passing JUnit test fails after executing a mutant, the mutant is considered killed.

PIT uses timeout to kill mutants that may never terminate. That is, if the execution of a mutant times out, then the mutant is considered killed. In order to reduce test execution time while preventing premature termination, we set the timeout value differently for each test set as follows. We first recorded the normal execution time taken by every test in a test set on the original program. This allowed us to find the longest execution time of a test set. If the longest execution time t is less than or equal to 10 seconds, we set the timeout value of the test set to be t plus 10 seconds. Otherwise, we set the timeout value to be t plus 100 seconds.

2.3 EXPERIMENTAL RESULTS

In this section, we present the results from our experiments. The coverage results for each algorithm are collected for the class files in the package that implement the algorithm, i.e., instead of every class file in the WEKA package. All the results and related files such as datafiles, scripts and experiment logs are publicly available at http://barbie.uta.edu/~hdfeng/.

2.3.1 IMPACT OF DATASETS

We executed each algorithm's default configuration with the 51 datasets. Some datasets are not applicable to a given algorithm, e.g., due to incorrect data type, insufficient number of attributes, and missing data of attributes. Table 2-1 (Section 2.2.2) shows the number of datasets that are applicable to each algorithm.

TABLE 2-5 BRANCH COVERAGE STATISTICS OF APPLICABLE DATASETS

	Apriori	**EM**	**J48**	**SimpleKMeans**	**LibSVM**
Maximum	28.79%	37.64%	36.64%	21.89%	34.96%
Minimum	26.72%	31.11%	8.33%	18.31%	20.46%
Mean	27.98%	35.34%	30.07%	21.04%	30.35%
Standard Deviation	0.68%	2.29%	4.8%	1.04%	3.77%
Median	28.02%	36.41%	29.89%	21.6%	31.23%

Table 2-5 presents some statistics about the branch coverage results of each selected algorithm with the applicable data sets. The results indicate that different datasets achieve similar coverage results despite significant differences in their sizes in terms of number of attributes and instances. Recall that as shown in Table 2-2 (Section 2.2.3), some datasets contain as many as 20,000 instances while other datasets contain as few as 14 instances. However, the standard deviations of the branch coverage results are generally less than 5% as shown in Table 2-5.

TABLE 2-6 APPLICABLE DATASETS FOR APRIORI

Dataset[a]	# of Attributes	# of Instances	Branch Coverage
vote	17	435	28.79%
weather.nominal	5	14	28.79%
splice	62	3190	28.62%
contact-lenses	5	24	28.28%
breast-cancer	10	286	28.28%
primary-tumor	18	339	27.76%
soybean	36	683	27.59%
supermarket	217	4627	27.59%
kr-vs-kp	37	3196	27.41%
mushroom	23	8124	26.72%

a. Only 10 datasets' branch coverage are available instead of 11 as shown in Table 2-1. Dataset *audiology* did not finish execution within 48 hours.

We point out that even some datasets have a very small number of instances, they can achieve higher branch coverage than the datasets with significantly more instances. Table VI shows the dataset and branch coverage information of the applicable datasets for the Apriori implementation. Consider dataset *weather.nominal* and *mushroom*. Dataset weather.nominal has only 5 attributes and 14 instances. Dataset mushroom has 23 attributes and 8124 instances. However, weather.nominal achieved higher coverage than mushroom. Similar situations exist for other algorithms, which are not shown due to space limitation.

Based on the results of the 51 datasets for each algorithm, we selected one dataset that achieved the highest branch coverage for each algorithm for the rest of our experiment. If more than one dataset achieves the highest branch coverage, we break the tie by choosing the one with

a smaller number of instances. For example, for Apriori, both vote and weather.nominal achieves the maximum branch coverage. To break the tie, we choose weather.nominal. The datasets selected for each algorithm are shown below

- Apriori – *weather .nominal*
- EM – *segment-challenge*
- J48 – *credit-a*
- SimpleKMeans – *iris.2D*
- LibSVM – *primary-tumor*

Finding 1: Larger datasets do not necessarily achieve higher branch coverage. In some cases, smaller datasets can achieve higher branch coverage than larger datasets.

Implication 1: The size of a dataset is not a dominating factor for determining test effectiveness of a dataset. Instead, other characteristics must be considered, e.g., the dataset structure, and the relationship between different data instances. Also, it is possible to create small datasets that are effective for testing data mining algorithms. number of dataset instances, when reducing input datasets or generating synthetic input datasets.

2.3.2 Branch Coverage Results of T-way Testing

Table 2-7 shows the branch coverage results of the seven test sets, including the negative test set, 1-way to 6-way positive test sets. Table 2-7 also shows the branch coverage results for the

default configuration as a baseline, and the branch coverage results that combine 6-way test and negative test.

TABLE 2-7 BRANCH COVERAGE RESULTS OF T-WAY TESTING

Test set	Apriori	EM	J48	SimpleKMeans	LibSVM
Default Configuration	28.79%	37.64%	36.64%	21.89%	34.96%
Negative	66.03%	50.54%	55.32%	66.24%	28.47%
1-way	55.52%	52.99%	52.73%	59.51%	24.47%
2-way	66.55%	53.80%	54.60%	69.53%	43.77%
3-way	68.62%	53.94%	59.77%	70.39%	54.63%
4-way	68.62%	53.94%	59.77%	70.39%	54.80%
5-way	68.62%	54.08%	59.77%	70.39%	54.80%
6-way	68.62%	54.08%	59.77%	70.39%	54.89%
6-way &Negative	68.97%	55.30%	59.77%	70.67%	55.34%

We observe that negative test sets achieve relatively high coverage, in comparison with positive t-way tests, for all the algorithms except LibSVM. One possible reason is that the validity of a configuration option value is not checked until it is used. Thus, in some cases, a significant amount of the source code could have been executed before the system detects this invalid value. We plan to investigate this further in our future work.

We also observe that the total coverage achieved by combining the negative test set and the 6-way test set ranges from 55.30% to 70.67%. Other empirical studies [4, 5, 26, 29, 43] have reported that 6-way test sets could detect all the faults. We plan to investigate this further in our

future work. The following factors could have contributed to the fact that our coverage results are less than expected:

- *Limited domain knowledge*: In our experiments, we performed input parameter modeling based on our limited domain knowledge. The input parameter models could be refined to achieve higher branch coverage.
- *Testing only configuration options*: In our experiments, CT is only applied to configuration options. That is, we did not test combinations between configuration options and different datasets.
- *Shared class files that contain unreachable code from implementations of other algorithms.* In WEKA, multiple algorithms are implemented within the same package. For example, the *weka.clusterers* package contains implementations of eight different clustering algorithms, e.g., SimpleKMeans, EM, Canopy, etc. Some portions of source code may only be reachable when executing its corresponding algorithm. As an example, SimpleKmeans algorithm is a variant of EM algorithm with the assumptions that clusters are spherical. In WEKA, EM algorithm uses the SimpleKMeans class to complete its first few steps of the clustering tasks. But most source code of SimpleKMeans are not semantically reachable because EM is only using a static configuration of SimpleKMeans algorithm as specific in the EM class file.

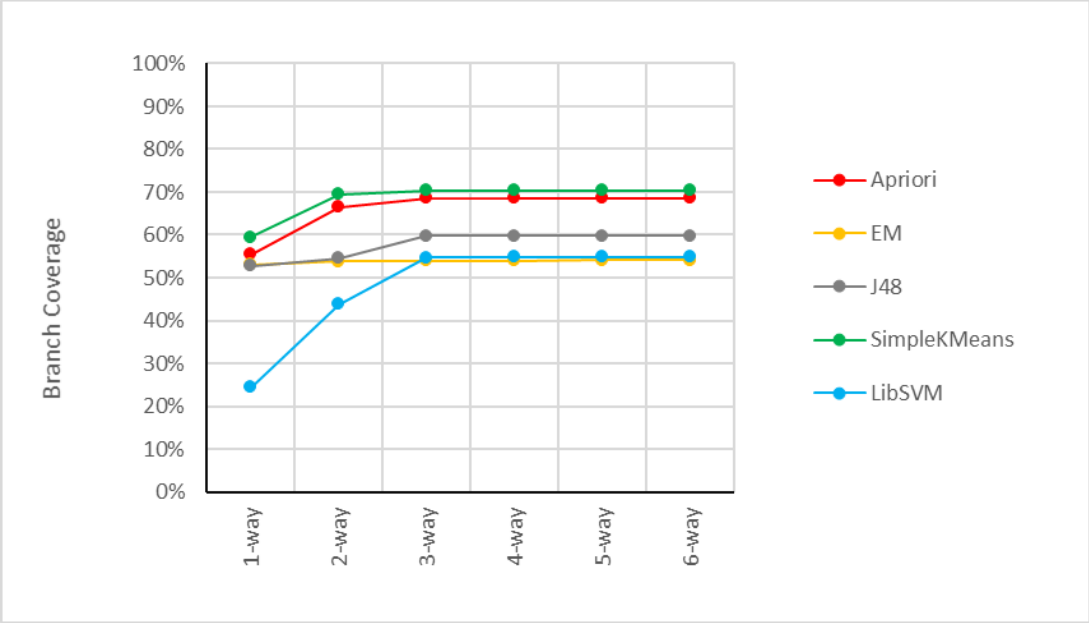


FIGURE 2-3 GROWTH OF BRANCH COVERAGE

Figure 2-3 shows how branch coverage increases with respect to test strength. The result is consistent with previous studies [15, 28]. That is, the coverage grows progressively slower when test strength increases. Also, branch coverage stops increasing after 3-way testing for algorithms Apriori, J48, SimpleKMeans, and after 5-way testing for algorithms EM. For algorithm LibSVM, branch coverage continues to increase until 6-way testing as shown in Table 2-7.

Finding 2: Branch coverage increases progressively slower as test strength increases. The coverage increase stops at a test strength that is relatively low.

Implication 2: During CT, data mining algorithms display similar behavior as general software applications. CT has the potential to be effective for testing data mining algorithms.

2.3.3 MUTATION COVERAGE RESULTS OF T-WAY TESTING

The mutation coverage results are unavailable for the following algorithms:

- KMeans – PIT cannot execute for this algorithm, due to a bug that is confirmed by PIT developers. Discussion of this bug is publicly available at https://github.com/hcoles/pitest/issues/300.
- EM – Mutation testing for EM was not able to complete within 48 hours due to the large number of mutators generated from the source code and the heavy computation of EM algorithm itself.

TABLE 2-8 MUTANTS GENERATED FOR EACH ALGORITHM

Mutants	**Apriori**		**J48**		**LibSVM**	
ConditionalsBoundaryMutator	120	4%	113	3.11%	314	6.38%
ConstructorCallMutator	186	6.2%	137	3.77%	140	2.85%
experimental	104	3.47%	202	5.56%	204	4.15%
IncrementsMutator	113	3.77%	91	2.5%	214	4.35%
InlineConstantMutator	555	18.5%	488	13.43%	876	17.81%
InvertNegsMutator	0	0%	1	0.03%	46	0.94%
MathMutator	99	3.3%	213	5.86%	511	10.39%
NegateConditionalsMutator	282	9.4%	348	9.57%	551	11.2%
NonVoidMethodCallMutator	778	25.93%	984	27.07%	614	12.48%
RemoveConditionalMutator	564	18.8%	696	19.15%	1102	22.41%
ReturnValsMutator	112	3.73%	239	6.57%	131	2.66%
VoidMethodCallMutator	87	2.9%	123	3.38%	215	4.37%
Total	3000	100%	3635	100%	4918	100%

Table 2-8 shows the number of mutants generated by each mutator. Some mutators are generating significantly more mutants than others as shown in Table 2-8

- *NonVoidMethodCallMutator*: Incorrect method calls.
- *InLineConstantMutator*: Assigning an incorrect constant value to a variable.
- *RemoveConditionalMutator*: Incorrect conditional statements. "RemoveConditionalMutator" will change the conditions to a constant boolean value.

For *NegateConditionalsMutator*, *RemoveConditional- Mutator* and *ConditionalBoundaryMutator*, these three mutators focus on generating mutants at conditional statements of the program, and these three mutators together generates over 30% of the total mutants for the three algorithms that are shown in Table 2-8.

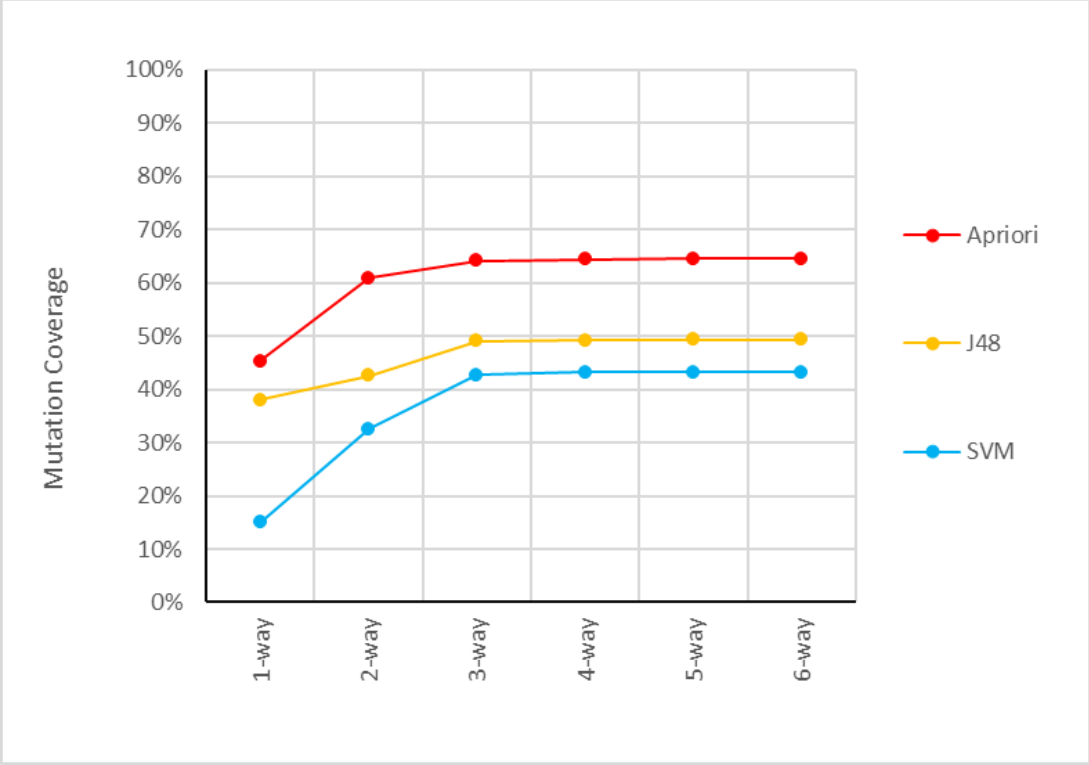


FIGURE 2-4 GROWTH OF MUTATION COVERAGE

Table 2-7 shows that branch coverage stops growing after 3-way testing for Apriori and J48. However, mutation coverage continues to grow for these two algorithms after 3- way testing, as shown in Table 2-9. This is because executing a line or branch of code does not necessarily expose faults that exist in the code or branch, especially when the computation or decision logic is more complex. However, the increase in mutation coverage is not significant after 3-way testing.

TABLE 2-9 MUTATION COVERAGE RESULTS OF T-WAY TESTING

Test set	Apriori	J48	LibSVM
Default Configuration	31.53%	29.66%	26.11%
Negative	59.90%	40.39%	19.46%
1-way	45.27%	38.05%	14.99%
2-way	60.93%	42.56%	32.61%
3-way	64.10%	49.05%	42.72%
4-way	64.47%	49.19%	43.19%
5-way	64.50%	49.35%	43.19%
6-way	64.53%	49.38%	43.21%
6-way&Negative	64.63%	49.38%	44.02%

Figure 2-4 plots the growth of mutation coverage with respect to test strength. The result is consistent with previous studies [15, 28] on branch coverage. Hence, Finding 2 and Implication 2 also apply to the coverage growth of mutation testing. The results of our experiment suggest that in general, branch coverage correlates well with mutation coverage. Thus, branch coverage could be used as a good indicator of fault detection effectiveness for data mining algorithms, since mutation coverage is expensive to measure.

When we analyzed the results of mutation coverage for the individual files of each algorithm, we discovered two special cases where mutation coverage for *"apriori.java"* for the Apriori algorithm decreased by one mutant from 4-way to 5- way and 6-way testing. As discussed in the Section II, the CT tests we created using ACTS used the *extend* mode. This means that every higher strength test set is a superset of its predecessor. Consequently, branch coverage and mutation coverage should not decrease from a lower strength test set to a higher strength test set. We have informed the lead developer of PIT with this issue, but the exact cause has not been successfully identified.

Finding 3: Branch coverage and mutation coverage seem to correlate well for data mining algorithms. That is, higher branch coverage seems to imply higher mutation coverage, and vice versa.

Implication 3: Branch coverage could be used as a good indicator of fault detection effectiveness for data mining algorithms, since mutation coverage is expensive to measure.

2.3.4 THREATS TO VALIDITY

Threats to external validity occur when the experimental results could not be generalized to other subjects. Our subject programs implement the top five data mining algorithms identified in [44] and are from a widely used data mining tool, i.e., WEKA. The datasets used in our experiments have been used in other studies [32]. More experiments using data mining algorithms

other than these five algorithms and using different datasets can further reduce threats to external validity.

Threats to internal validity are other factors that may be responsible for the experimental results. To prevent mistakes that could happen during the modeling process, two of the authors created the IPM independently and cross-checked them against each other. We have automated the execution of experiments using scripts, as an effort to minimize human errors. Furthermore, consistency of the results (executed by scripts) has been checked by two of the authors using their independently written scripts.

2.4. Related Work

We first review previous work on applying CT to different types of software. Lei et al. [30] developed a t-way testing strategy for testing concurrent programs. Simos et al. [40] and Bozic et al. applied CT to perform security testing of web applications [6]. Li et al. applied CT to test three real-life industrial software systems that include an embedded system, a graphical operating system and a database management system [35]. Dhadyalla et al. applied CT to test automotive control software embedded in a hybrid electric vehicle [16]. Li et al. applied CT to ETL applications [34]. Note that ETL is a special type of big data applications. However, the work in [34] focuses on data transformation and management aspects, whereas our work focuses on algorithmic aspects. These existing works show that CT can be effectively applied to different domains. However, to our knowledge, our work is the first one that applies CT to data mining algorithms.

Second, we review existing work related to evaluating the effectiveness of CT. Khun et al. [31] investigated the fault detection effectiveness of t-way testing. Kuhn et al. [27] report a study that applies CT and random testing to detect deadlocks in a network simulator. Bell and Vouk

discussed the effectiveness of pairwise testing and random testing to a network-centric software [2]. A number of studies have been reported that compares the effectiveness of CT and random testing [1, 5, 7, 17, 25, 41, 42]. There are also studies that investigate the code coverage effectiveness of t-way testing [13, 15]. Our work presented in this paper is the first effort to evaluate the effectiveness of CT to data mining algorithms.

Third, we review previous work related to testing data mining applications. Jeske et al. [24] developed a platform to generate realistic, synthetic data to test data mining tools. Data mining tools were evaluated in terms of their false positive and false negative error rates when executed with the synthetic data. Murphy et al. discussed how to identify metamorphic properties for performing metamorphic testing of data mining algorithms [38]. Metamorphic testing is one approach to addressing the test oracle problem. Murphy et al. [37] discussed approaches to test machine-learning applications that implement ranking algorithms. Our work is different in that we apply CT to test data mining algorithms.

Finally, we note that a significant amount of work has been reported on testing database centric applications [3, 8, 9, 10, 18, 33, 45]. Similar to work presented in [34], these work focuses on testing data management aspects. In contrast, our work focuses on the algorithmic aspects of data mining software.

2.5. Conclusion and Future Work

In this paper, we reported an experiment that applied CT to five data mining algorithms implemented in the WEKA tool. This is part of a larger effort that is aimed to develop effective CT-based methods for testing big data applications. The experiment allows us to obtain some initial understandings about the effectiveness of CT on data mining algorithms. In particular, the results of our experiment indicate that data mining algorithms behave in a way that is similar to

general software. This suggests that CT has the potential to be effectively applied to data mining algorithms.

We plan to continue our work in the following three directions. First, we will perform detailed code analysis to better understand the results of our experiment. In particular, we want to investigate why some branches were executed by none of our test sets, and whether these branches could be executed by using different configuration options and/or datasets. Second, in our experiment, we only applied CT to configuration options. We plan to investigate how to apply CT to create representative datasets. The key challenge is to identify the characteristics of a dataset that could significantly impact the execution of the underlying algorithm. We can model these characteristics as abstract parameters, and then apply CT to these parameters to create representative datasets. Third, negative testing alone has shown great importance in achieving good coverage, we will perform further investigation and experiments on how we can better use negative testing to improve the coverage of CT [20].

2.6. ACKNOWLEDGEMENT

This work is supported by a research grant (70NANB15H199) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

2.7. REFERENCES

[1] W. A. Ballance , S. Vilkomir, and W. Jenkins. Effectiveness of pair-wise testing for software with boolean inputs. Proceedings of the IEEE Fifth International Conference in Software Testing, Verification and Validation (ICST), 580-586, 2012.

[2] K.Z. Bell, and M.A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. Proceedings of the 3rd International Conference in Information and Communications Technology for Enabling Technologies for the New Knowledge Society, 221-235, 2005.

[3] C. Binnig, D. Kossmann, and E. Lo. Testing database applications. Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 739-741, 2006.

[4] M.N. Borazjany, L.S. Ghandehari, Y. Lei, R. Kacker and R. Kuhn. An input space modeling methodology for combinatorial testing. Proceedings of the Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 372-381, 2013.

[5] M.N. Borazjany, L. Yu, Y. Lei, R. Kacker and R. Kuhn. Combinatorial testing of ACTS: A case study. Proceedings of the Software Testing, Verification and Validation (ICST), 591-600, 2012.

[6] J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler, and F. Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. Proceedings of the IEEE International Conference In Software Quality, Reliability and Security (QRS), 207-212, 2015.

[7] R.C. Bryce, A. Rajan, and M.P. Heimdahl. Interaction testing in model-based development: Effect on model-coverage. Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC), 259-268, 2008.

[8] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos and E.J. Weyuker. A framework for testing database applications. Proceedings of the ACM SIGSOFT Software Engineering Notes, 25(5), 147-157, 2000.

[9] D. Chays, Y. Deng, P.G. Frankl, S. Dan, F.I. Vokolos and E.J. Weyuker. An AGENDA for testing relational database applications. Proceedings of the Software Testing, verification and reliability, 17-44, 2004.

[10] M.Y. Chan, and S.C. Cheung. Testing Database Applications with SQL Semantics. In CODAS, 363-374, 1999.

[11] M. Chen, S. Mao, and Y.Liu. Big data: A survey. Mobile Networks and Applications, 171-209, 2014.

[12] C.P. Chen and C.Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 275, 314-347, 2014.

[13] E.H. Choi, O. Mizuno, O and Y. Hu. Code Coverage Analysis of Combinatorial Testing. Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality, p.34.

[14] H. Coles. Pit mutation testing. http: //pitest.org/, 2016.

[15] J. Czerwonka. On use of coverage metrics in assessing effectiveness of combinatorial test designs. Proceedings of the IEEE Sixth International Conference in Software Testing, Verification and Validation Workshops (ICSTW), 257-266, 2013.

[16] G. Dhadyalla, N. Kumari and T. Snell. Combinatorial testing for an automotive hybrid electric vehicle control system: a case study. Proceedings of the IEEE Seventh International Conference in Software Testing, Verification and Validation Workshops (ICSTW), 51-57, 2014.

[17] M. Ellims, D. Ince and M. Petre. The effectiveness of t-way test data generation. Proceedings of the International Conference on Computer Safety, Reliability, and Security, 16-29, 2008.

[18] M. Emmi, R. Majumdar and K. Sen. Dynamic test input generation for database applications. Proceedings of the International Symposium on Software testing and analysis, 151-162, 2007.

[19] A. Frank, and A. Asuncion, A. UCI Machine Learning Repository [http://archive. ics. uci. edu/ml]. Irvine, CA: University of California. School of Information and Computer Science, 213, 2010

[20] A. Gargantini, J. Petke, M. Radavelli and P. Vavassori. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. Proceedings of the International Symposium on Search Based Software Engineering, 49-63, 2016.

[21] M. Grindal and J. Offutt. Input parameter modeling for combination strategies. Proceedings of the 25th conference on IASTED International Multi-Conference Software Engineering, 255-260, 2007.

[22] M. Hoffmann, B. Janiczak, E. Mandrikov and M. Friedenhagen. Jacoco code coverage tool. Online , 2016

[23] D.R. Jeske, P.J. Lin, C. Rendon, R. Xiao and B. Samadi. Synthetic data generation capabilties for testing data mining tools. Proceedings of the IEEE Military Communications conference(MILCOM), 1-6, 2006.

[24] D.R. Jeske, B. Samadi, P.J. Lin, L. Ye, S. Cox, R. Xiao et.al., Generation of synthetic data sets for evaluating the accuracy of knowledge discovery systems. Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, 756-762, 2005.

[25] N. Kobayashi, T. Tsuchiya and T. Kikuno. Applicability of non-specification-based approaches to logic testing for software. Proceedings of the IEEE International Conference on Dependable Systems and Networks, 337-346, 2001.

[26] R. Kuhn, R. Kacker, Y. Lei and J. Hunter. Combinatorial software testing. Computer, 42(8), 2009.

[27] D.R. Kuhn, R. Kacker and Y. Lei. Random vs. combinatorial methods for discrete event simulation of a grid computer network. Proceedings of ModSim World, 83-88, 2010.

[28] D.R. Kuhn, R. Kacker and Y. Lei. Introduction to CT. CRC press, 2013.

[29] D.R. Kuhn, R. Kacker and Y. Lei. Estimating t-Way Fault Profile Evolution During Testing. Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC) Vol. 2, 596-597, 2016.

[30] Y. Lei, R.H. Carver, R. Kacker and D.Kung. A combinatorial testing strategy for concurrent programs. Proceedings of the Software Testing, Verification and Reliability, 17(4), 207-225, 2007.

[31] D.R. Kuhn, D.R. Wallace and A.M. Gallo. Software fault interactions and implications for software testing. Proceedings of the IEEE transactions on software engineering, 30(6), 418-421, 2004.

[32] M. Lichman. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science, 2013.

[33] B. Li, M. Grechanik and D. Poshyvanyk. Sanitizing and minimizing databases for software application test outsourcing. Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation, 233-242, 2014.

[34] N. Li, Y. Lei, H.R. Khan, J. Liu and Y. Guo. Applying combinatorial test data generation to big data applications. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ACM) , 637-647, 2016.

[35] X. Li, R. Gao, W.E. Wong, C. Yang and D. Li. Applying Combinatorial Testing in Industrial Settings. Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 53-60, 2016.

[36] T.M. Mitchell, Machine learning. Burr Ridge, IL: McGraw Hill, 45(37), 870-877, 1989.

[37] C. Murphy, G.E. Kaiser and M. Arias. An Approach to Software Testing of Machine Learning Applications. In SEKE, 167, 2007.

[38] C. Murphy, G.E. Kaiser, L. Hu and L.Wu. Properties of Machine Learning Applications for Use in Metamorphic Testing. In SEKE (Vol. 8), 867-872, 2008.

[39] G. Paynter, L. Trigg, E. Frank and R. Kirkby. Attribute-relation file format (ARFF). Online] http://www. cs. waikato. ac. nz/ml/weka/arff. Html, 2008.

[40] D.E. Simos, K. Kleine, L.S.G. Ghandehari, B. Garn, and Y. Lei. A Combinatorial Approach to Analyzing Cross-Site Scripting (XSS) Vulnerabilities in Web Application Security Testing. Proceedings of the International Conference on Testing Software and Systems, 70-85, 2016.

[41] P.J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In Proceedings of International Symposium on Empirical Software Engineering, 49-59, 2004.

[42] S. Vilkomir, O. Starov and R. Bhambroo. Evaluation of t-wise approach for testing logical expressions in software. Proceedings of the IEEE Sixth International Conference in Software Testing, Verification and Validation Workshops (ICSTW), 249-256, 2013.

[43] H. Wang, C. Xu, J. Sui and J. Lu. How Effective Is Branch-Based Combinatorial Testing? An Exploratory Study. Proceedings of the IEEE International Conference in Software Quality, Reliability and Security (QRS), 41-52, 2016.

[44] X. Wu, V. Kumar, J.R. Quinlan, J. Ghosh, Q. Yang, H. Motoda et al. Top 10 algorithms in data mining. Knowledge and information systems, 14(1), 1-37, 2008.

[45] X. Wu, Y. Wang and Y. Zheng. Privacy preserving database application testing. Proceedings of the ACM workshop on Privacy in the electronic society, 118-128, 2003.

[46] L. Yu, Y. Lei, R.N. Kacker and D.R. Kuhn. Acts: A combinatorial test generation tool. Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 370-375, 2013.

Chapter 3. Effectiveness of dataset reduction in testing machine learning algorithms

The chapter contains a paper published in the IEEE 2nd International Conference on Artificial Intelligence Testing (AITest), in 2020.

Effectiveness of dataset reduction in testing machine learning algorithms*

Jaganmohan Chandrasekaran[1], Huadong Feng[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2]

[1]Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, USA

Abstract— Many machine learning algorithms examine large amounts of data to discover insights from hidden patterns. Testing these algorithms can be expensive and time- consuming. There is a need to speed up the testing process, especially in an agile development process, where testing is frequently performed. One approach is to replace big datasets with smaller datasets produced by random sampling. In this paper, we report a set of experiments that are designed to evaluate the effectiveness of using reduced datasets produced by random sampling for testing machine learning algorithms. In our experiments, we use as subject programs four supervised learning algorithms from the Waikato Environment for Knowledge Analysis (WEKA). We identify five datasets from Kaggle.com to run with the four learning algorithms. For each dataset, we generate reduced datasets of different sizes using two random sampling strategies, i.e., pure random and stratified random sampling. We execute our subject programs with the original and the reduced datasets, and measure test effectiveness using branch and mutation coverage. Our results indicate that in most cases, reduced datasets of even very small sizes can achieve the same or similar coverage achieved by the original dataset. Furthermore, our results indicate that reduced

datasets produced by the two sample strategies do not differ significantly, and branch coverage correlates with mutation coverage.

Keywords— Testing classifiers, Random sampling, Reduced datasets, Testing machine learning, Branch coverage, Software testing.

3.1 INTRODUCTION

Many machine learning algorithms examine large amounts of data to discover insights from hidden patterns. Given the nature of machine learning algorithms, testing can be expensive and time-consuming as each test case may have longer execution time compared to the testing of traditional applications. There is a need to speed up the testing process, especially in an agile development process, where testing is frequently performed. One approach is to replace high volume test datasets with smaller datasets produced by random sampling. One natural question to ask about this approach is the following: How does a reduced dataset compare to the original dataset in terms of effectiveness from a testing perspective?

In this paper, we investigate the effectiveness of using reduced datasets for testing machine learning algorithms. We measure test effectiveness using both branch coverage and mutation coverage. In our study, we use four supervised learning algorithms from the Waikato Environment for Knowledge Analysis (WEKA), which is a widely used machine learning workbench written in Java [1], as our subject programs. We identify five datasets, each of which represents a different application domain, from Kaggle.com to run with these algorithms. Kaggle.com is an online data science community that maintains a repository of public datasets.

After we identify subject programs and datasets, we first execute each subject program with each of the five datasets and measure test effectiveness in terms of branch coverage and mutation coverage. Second, we create two groups of reduced datasets. The first group is generated

using pure random sampling, i.e., in a purely random manner. The second group is generated using stratified random sampling, i.e., in a random manner that maintains the class distribution. In another word, a reduced dataset produced by stratified random sampling has the same class distribution as the original dataset. In the rest of the paper, we will refer to pure random sampling as random sampling and stratified random sampling as stratified sampling. Third, we execute the reduced datasets with subject programs and measure branch and mutation coverage. Finally, we compare the coverage results achieved by the reduced datasets to the coverage results achieved by the original datasets.

The major findings from our experiments are summarized as follows:

- In most cases, reduced datasets of even very small sizes achieve coverage identical or similar to the original datasets. In our experiments, the original datasets have the number of instances ranging from 142,193 to 999,999. The reduced datasets are of four sizes, i.e., 100, 200, 400, and 800, which are a fraction of the original dataset size. However, 522 out of 800 reduced datasets achieved the same coverage as the original datasets. Also, 112 out of 800 reduced datasets achieved more than 90% of the coverage achieved by the original datasets.
- One might expect that stratified sampling can be more effective than random sampling. However, in our experiments, the coverage results of the reduced datasets produced by the two sampling strategies are very similar. In particular, 628 out of 800 reduced datasets produced by the two sampling strategies achieved the same coverage. It is interesting to note that in several cases, random sampling achieved higher coverage than stratified sampling. The reason is that when the sample size is small, and when the dataset is skewed in terms of class distribution, stratified

sampling may produce no instances for a particular class, which could significantly reduce coverage.

- In most cases, branch coverage correlates with mutation coverage. Since mutation testing is quite expensive to perform, this suggests that branch coverage could be used as a practical alternative in place of mutation coverage for testing machine learning algorithms.

The rest of the paper is organized as follows. In Section 3.2 we present the design of our experiments, including the research questions, subject programs, datasets and metrics used in our experiments, and discussion about the generation of reduced datasets. Section 3.3 presents the results of our experiments, including branch and mutation coverage results for original and reduced datasets as well as implications of these results. Section 3.4 discusses potential threats to validity, including both internal and external threats. Section 3.5 reviews existing work that is related to ours. Section 3.6 provides conclusion remarks and a few directions for the future work.

3.2 Experimental Design

In this section, we present how we design our experiment, including the research questions, the selection of subject programs and datasets, the sampling approaches used to generate reduced datasets, and the metrics used to measure the effectiveness of the dataset executions.

3.2.1 Research Questions

Our main objective is to investigate the effectiveness of using a reduced dataset (in terms of volume, i.e., number of instances in a dataset) to test machine learning algorithms. We formulate the following research questions:

- How effective is it to test machine learning algorithms using reduced datasets, in comparison with the original datasets?
- How do the two sampling strategies, i.e., random sampling and stratified sampling, compare to each other?
- In testing machine learning algorithms, can branch coverage be used as a substitute for mutation coverage?

3.2.2 SUBJECT PROGRAMS

Waikato Environment for Knowledge Analysis (WEKA) is a machine learning workbench developed by University of Waikato. WEKA has a collection of supervised and unsupervised algorithms implemented in Java. Using WEKA, a user can perform tasks such as classification, regression, clustering and association rule mining. Four supervised algorithms from WEKA are used as our subject programs.

In WEKA, classification algorithms are categorized into seven different groups. We select one algorithm from each of the following four groups, bayes, meta, rules and trees. When we choose one algorithm from a group, we only consider algorithms that satisfy two conditions: (1) they support datasets with nominal class labels and (2) they generate a model at the end of its training phase. When there are multiple algorithms that satisfy the two conditions, we randomly choose one from these algorithms. The reason for condition (1) is that we use WEKA`s built-in filter to generate smaller datasets for stratified sampling. This filter is applicable only to datasets with nominal class labels. The reason for condition (2) is that during mutation testing, we need expected output to determine if a mutant is killed by comparing against the actual output. If an algorithm generates a model, then the model can be used as expected output during mutation testing.

For example, WEKA lists eight algorithms under the *trees* category. However, one of the eight algorithms, M5p, does not work on a nominal class label. Hence, we exclude M5P. Similarly, of the remaining seven algorithms, random forest works on a nominal class label dataset, but at the end of its training phase, the model is not accessible to the user with default configuration options. Hence, we exclude random forest. From the remaining six algorithms, we randomly select j48 as one of our subject algorithms.

Among different categories of classifiers listed in WEKA, we selected four algorithms namely NaiveBayes classifier [27], AdaBoost1 classifier [28], OneR classifier [29] and J48 classifier [30]. Table 3-1 lists our subject algorithms and some information about these algorithms, including package/class information, and number of branches and mutants. Each algorithm is executed with its default configuration values (as provided in WEKA) using command line interface (CLI).

Table 3-1 also lists information about an algorithm called DecisionStump. Classification accuracy of simple learning algorithms (weak learners), e.g., decision trees, naïve bayes, can be affected by potential bias in the training dataset. Thus, ensemble classifiers are used to improve their classification accuracy. AdaBoost1 belongs to a class of ensemble classifiers (boosting) that help to improve the classification accuracy of weak learners by training them iteratively, with different sets of weights assigned to class labels in each iteration. WEKA's default configuration of *AdaBoost1* implements a meta classifier that improves the accuracy of the model built using *DecisionStump*, a tree-based classifier (weak learner).

TABLE 3-1 INFORMATION ABOUT SUBJECT PROGRAMS

ALGORITHM	SUBJECT PROGRAMS	NUMBER OF BRANCHES	NUMBER OF MUTANTS
j48	weka.classifiers.trees.j48*	750	3796
NaiveBayes	weka.classifiers.bayes.NaiveBayes.java	203	1075
AdaBoost1	weka.classifiers.meta.AdaBoost1.java	90	491
DecisionStump	weka.classifiers.trees.DecisionStump.java	128	921
OneR	weka.classifiers.rules.OneR.java	88	510

3.2.3 DATASETS

We identify suitable datasets from Kaggle.com, which provides access to public databases. By default, dataset search results on Kaggle.com are sorted by hotness, a measure indicative of the amount of interests and recency of datasets on their platform [9]. Other methods of sorting include New, Recently Active, Most Votes, Updated and Relevance. As Kaggle.com does not release the hotness calculation formula to the public [10], we are not completely clear of how the hotness of datasets is computed. Hence, we sort the search results by Most Votes, which sorts datasets based on the most popular datasets of all time. Then, the results are further filtered with the following two criteria: (a) size – 10 MB to 1GB and (b) File types – CSV. Next, we inspect each dataset in the order sorted by Kaggle.com and select datasets that require no cleaning and can be executed in WEKA.

We identified five datasets from different application domains, including AustralianWeather [23], ForestCover [24, 25], Crime [26], SupplyChain [21] and VideoGames [22]. The ForestCover dataset is a multi-label classification dataset with seven different class labels. The remaining four datasets consist of binary class labels. Table II lists the datasets and their information.

We selected datasets such that data preprocessing is minimal. No modification was required for *AustralianWeather* and *SupplyChain* as their respective class labels were nominal by default. The class labels of the remaining three datasets, i.e. *ForestCover*, *Crime* and *VideoGames* were converted from numeric to nominal using WEKA's built-in filter.

TABLE 3-2 DATASET INFORMATION

DATASET	**# OF CLASS LABELS**	**# OF INSTANCES**	**# OF ATTRIBUTES**
ForestCover	7	581,012	55
AustralianWeather	2	142,193	23
Crime	2	284,807	31
SupplyChain	2	580,251	5
VideoGames	2	999,999	56

3.2.4 GENERATION OF REDUCED DATASETS

For each original dataset in Table 3-2, two groups of smaller datasets are generated. Group 1 consists of reduced datasets generated using pure random sampling, whereas in Group 2, reduced datasets are generated using stratified sampling. Recall that stratified sampling maintains the

overall class distribution of the original datasets. For each group, we generate samples of four different sizes, i.e., 100, 200, 400, 800. Also, in order to reduce variations in random sampling, we generate five samples for each sample size by using different random seeds. Thus, each dataset has 20 samples per group and a total of 40 samples in the two groups.

WEKA provides a set of pre-processing filters that allow users to modify datasets. Reduced datasets in Group 1 (random sampling) are generated using WEKA's pre- processing filter *weka.filters.unsupervised.instances*. Resample. Reduced datasets in Group 2 (stratified sampling) are generated using pre-processing filter *weka.filters.supervised.instances.Resample*. These filters allow the user to select the sample size, usually specified as a percentage of the original dataset. Note that both filters perform a volumetric reduction, i.e. the number of instances in the dataset is reduced whereas the number of attributes will remain unchanged.

For example, consider a dataset of 100,000 data instances with four class labels, A, B, C and D. Assume that their class distribution is as follows: 30% instances belong to Class A, 40% instances belong to Class B, 10% instances belong to Class C and the remaining 20% belongs to Class D. Generating a smaller dataset with 100 instances using stratified sampling (Group II) will consists of 30 instances belonging to Class A, 40 instances belonging to Class B, 10 instances belonging to Class C and 20 instances belonging to Class D. In contrast, samples generated using random sampling (Group I) does not necessarily maintain the class label distribution.

The Crime dataset (284,807 instances) has the following class distribution: 99.82% instances belong to Class 0 (284,315 instances), and 0.18% instances belong to Class 1 (492 instances). When generating a reduced dataset with 800 instances using WEKA's pre-processing filter, it is highly likely that random sampling fails to produce a reduced dataset that include instances in both Class 0 and Class 1. Instead, it is likely that all of the 800 instances belong to

Class 0. A developer might face the above said scenario when s/he generates a reduced dataset using random sampling from a class-imbalanced (or skewed) dataset. As a workaround, a developer can create a reduced dataset while preserving the original class distribution. This is our motivation to use two different groups of samples and to investigate their impact in testing supervised learning algorithms. The original datasets and their reduced versions are made publicly available at [32].

3.2.5 METRICS

We use both branch coverage and mutation coverage to measure test effectiveness. Branch coverage is recorded using JaCoCo [18]. We choose branch coverage over statement coverage because the former subsumes the latter. We note that logic coverage is stronger than branch coverage. Unfortunately, JaCoCo does not report logic coverage.

Mutation coverage is obtained using PITest (PIT), which is a widely used mutation testing framework [19]. PIT can automatically seed one fault at a time into SUT and execute the mutated code against the unit test(s) specified. We executed each dataset with WEKA's default configuration options and the output (model) is saved in a .txt file (expected output). Then, we used jUnit tests to compare the expected output against the output of each mutated version. If the jUnit tests fail on execution, the mutant is considered killed. In our experiments, we have thirteen mutation operators including all the default mutators (seven), three experimental mutators and three optional mutators [20, 31].

The machine we used for our experiments is a workstation with two Xeon E5- 2630V3 8 core CPUs @ 2.40GHz, 64GB DDR4 2133 MT/s memory, and a Samsung 850 EVO 500GB SSD.

3.3 Experimental Results

In this section, we present our experimental results and discussion about our results. In Section 3.3.1, we present the branch coverage results achieved by the original datasets. These results are considered to be the baseline results. In Section 3.3.2, we present the branch coverage results achieved by the reduced datasets. These results are compared to the baseline results. In Section 3.3.3, we present the mutation coverage results achieved by both of the original and reduced datasets.

3.3.1 Branch Coverage of the Original Datasets

Table 3-3 presents the branch coverage achieved by algorithms with original datasets. Among the datasets, *SupplyChain* consistently achieve higher coverage for all the algorithms. We observe that across algorithms, a considerable number of methods, and their branches were not executed, and thus the overall branch coverage appears to be considerably lower (<= 50%). This, however, can be explained as follows. Consider the branch coverage results of the OneR algorithm. The *SupplyChain* dataset achieves the highest branch coverage (57%), i.e., 51 out of 88 total branches. Among the missing 37 branches, 18 branches missed due to default configuration options. Seven branches are related to error handling, such as missing attribute values, and the remaining 12 branches cannot be covered as cross-validation is not performed while building models using the command-line interface (CLI).

To our surprise, *AustralianWeather* covers a significantly smaller number of branches (17) compared to the rest. This can be explained as follows: Among the five datasets, all the attributes of *AustralianWeather* belong to the nominal data type. All the attributes of *ForestCover*, *VideoGames*, and *Crime* belong to the numeric data type. In the case of *SupplyChain*, 3 out of 4 attributes belong to the numeric data type, and the remaining attribute belongs to the nominal data

type. When executing the *OneR* algorithm with *AustralianWeather*, a method, newNumericRule(), was missed that has 36 branches and handles numeric attributes. Hence, *AustralianWeather* achieves a significantly lower branch coverage, whereas *SupplyChain* achieves the highest branch coverage, as it covers branches related to both numeric and nominal data types.

In our experiments, we executed the algorithms using WEKA's default configuration options only. This could cause branching conditions that are specific for other configuration options to be missed. As shown in [2], executing different configuration options could significantly increase branch coverage. Also, the branches related to error handling and GUI are not covered as we run our tests with clean datasets using the CLI.

We emphasize that, although branch coverage achieved by original datasets is not high, this does not affect the purpose of our experiments, which is to determine whether reduced datasets could achieve the same or similar coverage as the original dataset.

TABLE 3-3 BRANCH COVERAGE FOR ORIGINAL DATASETS

DATASETS	ALGORITHMS	# OF BRANCHES COVERED	TOTAL NUMBER OF BRANCHES	BRANCH COVERAGE
AustralianWeather	j48	180	750	24%
ForestCover		202		26%
SupplyChain		201		26%
VideoGames		202		26%
Crime		195		26%
AustralianWeather	Naïve Bayes	73	203	35%
ForestCover		77		37%
SupplyChain		99		48%
VideoGames		79		38%
Crime		78		38%
AustralianWeather	AdaBoost1	28	90	31%
ForestCover		17		18%
SupplyChain		28		31%
VideoGames		28		31%
Crime		28		31%
AustralianWeather	DecisionStump	50	128	39%
ForestCover		47		36%
SupplyChain		71		55%
VideoGames		48		37%
Crime		48		37%
AustralianWeather	OneR	17	88	19%
ForestCover		44		50%
SupplyChain		51		57%
VideoGames		45		51%
Crime		45		51%

TABLE 3-4 RELATIVE BRANCH COVERAGE OF REDUCED DATASETS (RANDOM SAMPLING)

DATASETS	ALGORITHMS	SIZE OF THE REDUCED DATASET			
		100	200	400	800
AustralianWeather	j48	0.75	0.75	0.71	0.71
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		0.81	0.73	0.92	1.00
VideoGames		0.96	0.96	0.96	1.00
Crime		0.12	0.35	0.12	0.35
AustralianWeather	Naïve Bayes	1.00	1.00	1.00	1.00
ForestCover		1.03	1.03	1.03	1.03
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		1.00	1.00	1.00	1.00
AustralianWeather	AdaBoost1	1.00	1.00	1.00	1.00
ForestCover		1.78	1.67	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.65	0.65	0.65	0.65
AustralianWeather	DecisionStump	0.95	0.95	0.95	0.95
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.97	1.00	0.97	1.00
AustralianWeather	OneR	0.95	0.95	0.95	0.95
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		0.96	0.96	0.96	0.96
VideoGames		1.00	1.00	1.00	1.00
Crime		0.76	0.92	0.76	1.00

TABLE 3-5 RELATIVE BRANCH COVERAGE OF REDUCED DATASETS (STRATIFIED SAMPLING)

DATASETS	ALGORITHMS	SIZE OF THE REDUCED DATASET			
		100	200	400	800
AustralianWeather	j48	0.75	0.75	0.71	0.71
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		0.81	0.92	0.96	1.00
VideoGames		0.92	0.96	1.00	1.00
Crime		0.12	0.12	0.12	0.35
AustralianWeather	Naïve Bayes	1.00	1.00	1.00	1.00
ForestCover		1.03	1.03	1.03	1.03
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		1.00	1.00	1.00	1.00
AustralianWeather	AdaBoost1	1.00	1.00	1.00	1.00
ForestCover		1.78	1.67	1.78	1.67
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.65	0.65	0.65	1.00
AustralianWeather	DecisionStump	1.00	1.00	1.00	1.00
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.97	0.97	0.97	1.00
AustralianWeather	OneR	0.95	0.95	0.95	0.95
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		0.96	0.96	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.76	0.76	0.76	0.92

3.3.2 Branch Coverage of Reduced Datasets

In this section, we present the branch coverage results achieved by reduced datasets. For each dataset, we generate reduced datasets using two different approaches: random sampling and stratified sampling; we generate reduced datasets in four different sizes: 100 instances, 200 instances, 400 instances, and 800 instances, as discussed in Section 3.2.4. Due to limited space, we present the median branch coverage achieved by each size relative to their baseline coverage.

Tables 3-4 and 3-5 present the branch coverage results of reduced datasets generated using random sampling and stratified sampling, respectively. All the coverage results presented here are relative to their corresponding baseline. i.e., a relative branch coverage of 1.0 suggests that a reduced dataset achieves a branch coverage identical to the original dataset. Note that, in Tables 3-4 and 3-5, 39 out of 50 reduced datasets of size 800 produced by both random and stratified sampling, achieved branch coverages identical to the baseline; for the remainder of the cases, we notice the coverages do not significantly vary among different sample sizes. Therefore, in our experiments we did not consider sample size larger than 800 instances.

The results indicate that, for the j48 algorithm, reduced datasets of size 800 instances produced by both random and stratified sampling of ForestCover, SupplyChain, and VideoGames can retain their baseline branch coverage. For the NaiveBayes algorithm, the reduced versions of all five datasets can retain their branch coverage achieved by their respective original datasets and in some cases, reduced datasets achieving even higher branch coverage. Similarly, for the remaining three algorithms namely AdaBoost1, DecisionStump, and OneR, the reduced versions of all datasets except Crime, in most cases either retain their respective baseline branch coverage (1.0) or in some cases achieve a branch coverage closer to its baseline (0.9<=branch coverage<1.0).

For the reduced datasets of *Crime*, we observe that three out of five algorithms (j48, AdaBoost1, One-R) suffer from a loss in branch coverage. In particular, consider the case of j48 (Row 5 in Tables 3-4 and 3-5), which suffers from a significant loss in branch coverage. This is attributed to the class imbalance problem. The Crime dataset consists of 284,807 instances with two class labels: (0, 1); 99.82% instances belonging to Class 0 and remaining 0.18% belonging to Class 1. Due to class imbalance, chances of drawing all hundred samples (at random) that belong to Class 0 is higher.

In our experiments, for the reduced datasets of size 100 produced by random sampling, four out of five samples have all their instances belonging to Class 0, and they achieve a relative median branch coverage of 0.12. On the contrary, three out of five reduced datasets of size 200 produced by random sampling have representation from both of the class labels, and they achieve a higher branch coverage comparatively (0.35). We notice that, in the case of j48, if a reduced dataset consists of a single label, there is a significant loss in branch coverage.

Next, we compare the coverage results of random sampling and stratified sampling. Our results indicate that, in most cases, the datasets reduced using both random and stratified sampling can achieve the same branch coverage.

In the cases of *AustralianWeather*, *SupplyChain* and *VideoGames*, the datasets reduced using both random and stratified sampling achieves identical branch coverage. This can be explained by the fact that all reduced datasets have a good class label representation. For example, all five sample datasets of AustralianWeather of size 100 that are reduced using stratified sampling have the following class label distribution: 78 instances belong to No, and 22 instances belong to Yes. In the case of random sampling, amongst five samples, sample 5 consists of 86 instances

belong to No and 14 instances belongs to Yes whereas, Sample 3 consists of 74 instances belong to No and 26 instances belongs to Yes.

Our results indicate that the reduced datasets of ForestCover generated using both random and stratified sampling achieve the same branch coverage as the original datasets across all algorithms. In comparison, the reduced datasets generated from *AustralianWeather*, *SupplyChain*, *VideoGames*, and *Crime* suffer from a minimal to moderate coverage loss in at least one of the five algorithms. This may be attributed to the fact, *ForestCover* is a multilabel dataset (7 class labels), whereas the rest of the four datasets are binary label dataset. More experimental data is required to obtain a better understanding. Also, our results indicate that in the case of the AdaBoost1 algorithm, the reduced datasets achieve a better branch coverage compared to the baseline, i.e., the original datasets. To some extent, this result is surprising, given the significant increase in branch coverage. This is possible because the reduced datasets may trigger execution scenarios that are different than the original datasets.

In the case of the *Crime* dataset, three algorithms suffer from a coverage loss. In particular, consider the coverage achieved by the reduced datasets of *Crime* produced by both random and stratified sampling. Row 5 in Tables 3-4 and 3-5 indicates that the reduced dataset of size 200 produced by random sampling achieves a higher branch coverage (0.35) compared to the reduced dataset produced by stratified sampling of the same size (0.12). This can be attributed to the representativeness of the class label. On examination of reduced datasets, we observe that three out of five samples generated using random sampling have instances belonging to two class labels (Class 0 and Class 1). However, in the case of datasets reduced using stratified sampling, all instances belong to a single class (Class 0). Hence, subject programs achieve lower coverage while executing with stratified samples as they fail to trigger the execution of certain branches. The

branch coverage results of the OneR algorithm suggest a similar pattern, i.e., the reduced dataset of size 200 produced by random sampling achieves a higher coverage (0.92) compared dataset reduced using stratified sampling of the same size (0.76).

This behavior of stratified sampling, i.e., all the instances of a reduced dataset belonging to a single class, is expected as it draws samples in a way that maintains the class distribution of the original dataset. Recall that the Crime dataset consists of 284,807 instances with two class labels: (0, 1); 99.82% instances belonging to Class 0 and remaining 0.18% belonging to Class 1. To generate a reduced dataset of size 200 instances using stratified sampling, instances are drawn in the following way (99.82% * 200) > 199 (instances) belonging to Class 0 and (0.18% * 200) < 1 (instances) belonging to Class 1. Hence, all the instances belong to Class 0 and thus, the reduced dataset suffers from lack of class representativeness.

For the Crime dataset, a minimum of 556 instances is required to guarantee that a reduced dataset (stratified sampling) consists of instances belonging to both classes (0 and 1). Among four different sizes (100, 200, 400, and 800) of reduced datasets generated using stratified sampling, in three groups (100,200 and 400), all instances belong to class 0 and thus achieve a low branch coverage (0.12). In the case of reduced datasets of 800 instances, all five samples consist of instances of both classes and thus achieve a relatively higher branch coverage (0.35).

Our results indicate that approximately 80% of the reduced datasets achieve coverage identical or similar to the original datasets. In another word, the volume of a dataset does not directly attribute to branch coverage. Instead, factors such as lack of representativeness of class labels in a reduced dataset could impact branch coverage. The results suggest that in most cases, reduced datasets do not suffer from branch coverage loss. In this respect, they can be used in place of the original datasets to speed up the testing process.

Among the two sampling approaches, the results indicate that in most cases (around 75%) reduced datasets generated using both random and stratified sampling exhibit identical behavior. However, when a tester decides to use stratified sampling, he/she should choose the size of the reduced dataset (minimum number of samples) based on the original class distribution such that each class label is represented in the reduced dataset.

3.3.3 MUTATION COVERAGE OF REDUCED DATASETS

In this section, we present the mutation coverage results achieved by algorithms while executing with reduced datasets.

Given the size of the datasets and the number of mutants generated for SUT, the overall execution time can be between a few hours to several days. Due to time constraints, our experiments have an execution time limit of 48 hours (chosen arbitrarily). If a dataset takes more than 48 hours to complete, then we kill the test execution and use a relatively smaller dataset (10000 instances) as our baseline. Out of 20 baseline test executions, one baseline execution, *j48 algorithm* with the *VideoGames* dataset executed for more than 2 days. Hence, we generated five smaller samples of *VideoGames* dataset with 10000 instances each and used their median coverage as a baseline.

Tables 3-6 and 3-7 present the mutation coverage results of the reduced datasets. All the coverage results presented here are relative to their corresponding baseline. The results from Tables 3-6 and 3-7 suggest that the j48 algorithm performs poorly with the reduced datasets of *AustralianWeather* and *SupplyChain*. Similarly, the reduced datasets of *Crime* result in a mutation coverage decrease for all the algorithms except Naive Bayes. The rest of the reduced datasets generated using both random and stratified sampling can retain their baseline mutation coverage.

TABLE 3-6 RELATIVE MUTATION COVERAGE OF REDUCED DATASETS (RANDOM SAMPLING)

DATASETS	ALGORITHMS	SIZE OF THE REDUCED DATASET			
		100	200	400	800
AustralianWeather	j48	0.50	0.50	0.44	0.50
ForestCover		0.96	0.96	0.96	1.00
SupplyChain		0.64	0.57	0.71	0.79
VideoGames		0.88	0.88	0.92	0.96
Crime		0.14	0.24	0.14	0.24
AustralianWeather	Naïve Bayes	0.94	0.94	0.94	0.94
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		1.00	1.00	1.00	1.00
AustralianWeather	AdaBoost1	1.00	1.00	1.00	1.00
ForestCover		1.92	1.38	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.04	1.00	1.00	1.00
Crime		0.50	0.54	0.50	0.54
AustralianWeather	DecisionStump	1.00	1.00	1.00	1.00
ForestCover		1.03	1.00	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.85	0.94	0.85	0.94
AustralianWeather	OneR	0.93	0.93	0.93	0.93
ForestCover		0.97	1.00	1.00	1.00
SupplyChain		1.07	1.07	1.07	1.07
VideoGames		0.97	0.97	1.00	1.00
Crime		0.66	0.77	0.66	0.89

TABLE 3-7 RELATIVE MUTATION COVERAGE OF REDUCED DATASETS (STRATIFIED SAMPLING)

DATASETS	ALGORITHMS	SIZE OF THE REDUCED DATASET			
		100	**200**	**400**	**800**
AustralianWeather	j48	0.50	0.50	0.50	0.50
ForestCover		0.92	0.96	1.00	0.96
SupplyChain		0.64	0.71	0.79	0.79
VideoGames		0.54	0.88	0.96	0.96
Crime		0.14	0.14	0.14	0.24
AustralianWeather	Naïve Bayes	0.94	0.94	0.94	0.94
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		1.00	1.00	1.00	1.00
AustralianWeather	AdaBoost1	1.00	1.00	1.00	1.00
ForestCover		2.00	1.38	2.00	1.38
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.50	0.50	0.50	1.00
AustralianWeather	DecisionStump	1.00	1.00	1.00	1.00
ForestCover		1.03	1.00	1.03	1.00
SupplyChain		1.00	1.00	1.00	1.00
VideoGames		1.00	1.00	1.00	1.00
Crime		0.85	0.85	0.85	0.97
AustralianWeather	OneR	0.93	0.93	0.93	0.93
ForestCover		1.00	1.00	1.00	1.00
SupplyChain		1.07	1.07	1.13	1.07
VideoGames		0.97	1.00	0.97	1.00
Crime		0.66	0.66	0.66	0.77

We report that the majority of the mutation coverage results (except reduced datasets of *SupplyChain* on j48) mirrors with their respective branch coverage results (Table 3-4 & 3-5; Table 3-6 & 3-7). Figures 3-1 and 3-2 present a correlation graph of branch coverage vs. mutation coverage for random sampling and stratified sampling, respectively. In Figures 3-1 and 3-2, x-axis indicates branch coverage, and the y-axis indicates mutation coverage. For the datasets reduced via random sampling, branch vs. mutation coverage has a Pearson correlation coefficient of 0.944148, whereas the datasets reduced via stratified sampling has a fractionally lower Pearson correlation coefficient of 0.939506. The result suggests that in most cases, mutation coverage has a strong positive correlation with the branch coverage. To our surprise, the mutation results of j48 using the *SupplyChain* dataset reduced using stratified sampling does not appear to correlate well with branch coverage, and we plan to investigate this further as part of our future work.

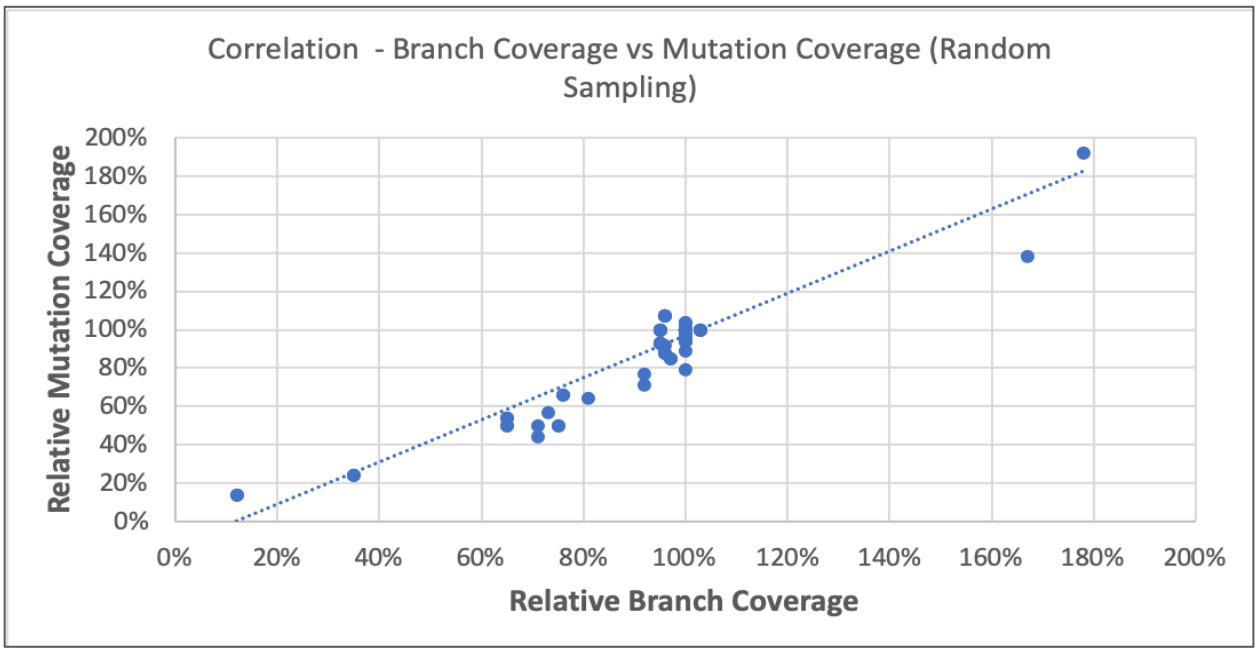


FIGURE 3-1 CORRELATION GRAPH – RANDOM SAMPLING

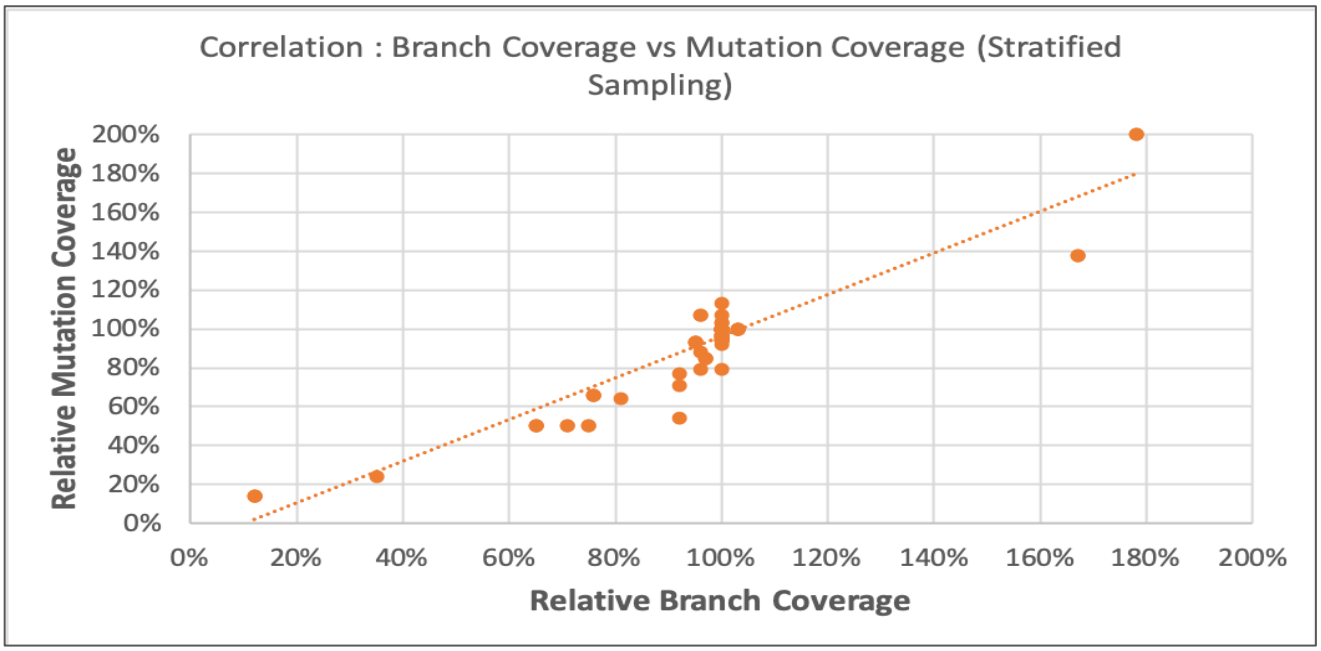


FIGURE 3-2 CORRELATION GRAPH – STRATIFIED SAMPLING

3.4. THREATS TO VALIDITY

Threats to internal validity are factors that may be responsible for experimental results, without our knowledge. To reduce human errors in the experimental procedure, we tried to automate our experiments as much as possible. In particular, we wrote scripts to automatically execute tests, measure code and mutation coverage, and generate coverage reports. Further, the results generated from samples of each dataset were verified manually, whenever possible.

Threats to external validity occur when the experimental results could not be generalized to other subjects. Using a single dataset for our experiments might impact the validity of our results due to lack of representativeness. To mitigate this threat, we used four supervised learning algorithms from WEKA that belong to different groups and five datasets from different application

domains. More experiments using other learning algorithms, including both supervised and unsupervised algorithms, and other datasets, can further reduce the threats to external validity.

3.5. Related Work

First, we review existing work reported on testing machine learning algorithms. One challenge in testing machine learning algorithms is how to deal with the test oracle problem. Murphy et al. [4,5] proposed a metamorphic testing technique to test machine learning algorithms. They developed metamorphic properties for three machine learning algorithms, including MartiRank, SVMLight, and PAYL. Similarly, Nakajima et al. [7] proposed a systematic approach to derive metamorphic properties and translation functions for testing a special class of classifiers known as Support Vector Machines (SVM). Xie et al. [11] proposed a metamorphic testing approach to test supervised learning algorithms, namely Naïve Bayes classifier and k-nearest neighbor classifier. Our work differs from these works in that we focus on evaluating the effectiveness of using smaller datasets in testing supervised learning algorithms.

Next, we review existing work on dataset reduction for big data applications [3, 6, 8, 13, 14, 15, 16, 17]. Such work is relevant because many machine learning algorithms are big data applications in that they are designed to learn from large amounts of data. Ur Rehman et al. [13] reviewed existing data reduction techniques such as compression- based data reduction method, dimension reduction techniques for big data applications. Czarnowski et al. [14] proposed an agent-based population learning algorithm for data reduction. Their algorithm aims at finding a subset of the original dataset that can be used to build a classifier that is similar to the classifier built using the original dataset. In contrast, our work focuses on volume reduction and its impact on test effectiveness.

Rojas et al. [38] investigate how different sampling strategies could impact data exploration on big datasets by comparing the performance of smaller datasets generated using random sampling and three non-random sampling techniques namely Query by committee, Density, and Uncertainty sampling. These works try to discover the same amount of information with a reduced dataset, which is different from our work, which tries to find a subset of the original dataset that preserves test effectiveness. To the best of our knowledge, our work is the first to investigate the effectiveness of dataset reduction in testing machine learning algorithms.

Finally, we mention that there are studies in the literature that investigate the effect of test suite minimization on fault detection effectiveness [33, 34, 35, 36, 37]. A test suite is different than a dataset, as the former is a set of test cases each of which represents an independent test input, whereas the latter is a set of instances that are together used as one single test input.

3.6. Conclusion and Future Work

In this paper, we report a study that investigates the use of reduced datasets in testing machine learning algorithms. We used four supervised learning algorithms from WEKA as our subject programs. Five publicly available datasets from Kaggle.com were chosen as subject datasets. For each dataset, we generated reduced datasets in four different sizes using random and stratified sampling. Then, we executed the algorithms with the original and the reduced datasets and measured test effectiveness in terms of branch and mutation coverage. Our results indicate, in most cases, reduced datasets of very small sizes (e.g. 800 instances) can retain branch and mutation coverage of the original, big datasets (e.g., >100,000 instances). This suggests that reduced datasets can be used to effectively test machine learning algorithms. Our results also indicate a high correlation between branch coverage and mutation coverage. Thus, branch coverage can be used when mutation testing is prohibitively expensive.

This is the first step in our larger effort to speed up testing machine learning algorithms. We plan to continue our work in the following directions. First, we plan to investigate the reduction of even bigger multi-label datasets (> 1 GB) and its effect on testing machine learning algorithms. Second, we plan to expand our study to include unsupervised learning algorithms. Compared to supervised learning algorithms, unsupervised learning algorithms learn from unlabeled datasets and thus could be harder to validate its output. Third, our experiments show that there exists a high correlation between branch and mutation coverage. However, some recent work reports that traditional code coverage measures such as branch coverage may not be adequate for testing deep learning algorithms. We believe that this has to do with the nature of the algorithms and also the types of fault that may exist in the algorithms. We plan to study this further by conducting experiments on deep learning algorithms. Finally, we plan to develop new methods, i.e., methods other than random sampling, for dataset reduction. For example, how to perform equivalence partitioning among instances in a big dataset, and then choose one or more representatives from each equivalence group.

3.7. ACKNOWLEDGEMENT

This work is supported by research grant (70NANB18H207) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

3.8. REFERENCES

[1] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten (2009). The WEKA Data Mining Software: An Update. SIGKDD Explorations, Volume 11, Issue 1.

[2] Chandrasekaran, Jaganmohan, et al. "Applying combinatorial testing to data mining algorithms." 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2017.

[3] Feldman, Dan, Melanie Schmidt, and Christian Sohler. "Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering." Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2013.

[4] Murphy, Christian, Gail E. Kaiser, and Marta Arias. "An approach to software testing of machine learning applications." (2007).

[5] Murphy, Christian, Gail E. Kaiser, and Lifeng Hu. "Properties of machine learning applications for use in metamorphic testing." (2008).

[6] Kira, Kenji, and Larry A. Rendell. "The feature selection problem: Traditional methods and a new algorithm." Aaai. Vol. 2. 1992.

[7] Nakajima, Shin, and Hai Ngoc Bui. "Dataset coverage for testing machine learning computer programs." 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2016.

[8] Khalid, Samina, Tehmina Khalil, and Shamila Nasreen. "A survey of feature selection and feature extraction techniques in machine learning." 2014 Science and Information Conference. IEEE, 2014.

[9] Datasets Documentation, https://www.kaggle.com/docs/datasets.

[10] Hotness calculation formula, https://www.kaggle.com/general/39290

[11] Xie, Xiaoyuan, et al. "Testing and validating machine learning classifiers by metamorphic testing." Journal of Systems and Software 84.4 (2011): 544-558.

[12] Zhang, Zhiyi, and Xiaoyuan Xie. "Towards testing big data analytics software: the essential role of metamorphic testing." Biophysical reviews 11.1 (2019): 123-125.

[13] ur Rehman, Muhammad Habib, et al. "Big data reduction methods: a survey." Data Science and Engineering 1.4 (2016): 265-284.

[14] Czarnowski, Ireneusz, and Piotr Jędrzejowicz. "An Approach to Data Reduction for Learning from Big Datasets: Integrating Stacking, Rotation, and Agent Population Learning Techniques." Complexity 2018 (2018).

[15] Czarnowski, Ireneusz, and Piotr Jędrzejowicz. "Stacking and rotation-based technique for machine learning classification with data reduction." 2017 IEEE International Conference on INnovations in Intelligent SysTems and Applications (INISTA). IEEE, 2017.

[16] Liu, Qingzhong, et al. "Mining the big data: The critical feature dimension problem." 2014 IIAI 3rd International Conference on Advanced Applied Informatics. IEEE, 2014.

[17] Wold, Svante, Kim Esbensen, and Paul Geladi. "Principal component analysis." Chemometrics and intelligent laboratory systems 2.1-3 (1987): 37-52.

[18] M. Hoffmann, B. Janiczak, E. Mandrikov and M. Friedenhagen. Jacoco code coverage tool. Online , 2016

[19] H. Coles. Pit mutation testing. http: //pitest.org/, 2016.

[20] Coles, Henry, et al. "Pit: a practical mutation testing tool for java." Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016.

[21] LEGO Database, https://www.kaggle.com/rtatman/lego-database#inventory_parts.csv

[22] League of Legends Ranked Matches, https://www.kaggle.com/paololol/league-of-legends-ranked-matches#stats1.csv

[23] Rain in Australia, https://www.kaggle.com/jsphyg/weather-dataset-rattle-package

[24] As Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science

[25] Forest Cover Type Prediction, https://www.kaggle.com/c/forest-cover-type-prediction/overview

[26] Credit Card Fraud Detection, https://www.kaggle.com/mlg-ulb/creditcardfraud

[27] John, George H., and Pat Langley. "Estimating continuous distributions in Bayesian classifiers." Proceedings of the Eleventh conference on Uncertainty in artificial intelligence. Morgan Kaufmann Publishers Inc., 1995.

[28] Freund, Yoav, and Robert E. Schapire. "Experiments with a new boosting algorithm." icml. Vol. 96. 1996.

[29] Holte, Robert C. "Very simple classification rules perform well on most commonly used datasets." Machine learning 11.1 (1993): 63-90.

[30] Salzberg, Steven L. "C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993." Machine Learning 16.3 (1994): 235-240.

[31] Mutation Operators, https://pitest.org/quickstart/mutators/

[32] OneDrive, https://1drv.ms/f/s!AjZ3W-Mz9wPKhtlLoWUU2zZKzm4bRg

[33] Wong, W. Eric, et al. "Effect of test set minimization on fault detection effectiveness." Software: Practice and Experience28.4 (1998): 347-369.

[34] Wong, W. Eric, et al. "Test set size minimization and fault detection effectiveness: A case study in a space application." Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97). IEEE, 1997.

[35] Rothermel, Gregg, et al. "An empirical study of the effects of minimization on the fault detection capabilities of test suites." Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, 1998.

[36] Jones, James A., and Mary Jean Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage." IEEE Transactions on software Engineering 29.3 (2003): 195-209.

[37] Rothermel, Gregg, et al. "Empirical studies of test-suite reduction." Software Testing, Verification and Reliability 12.4 (2002): 219-249.

[38] Rojas, Julian A. Ramos, et al. "Sampling techniques to improve big data exploration." *2017 IEEE 7th symposium on large data analysis and visualization (LDAV)*. IEEE, 2017.

Chapter 4. A Combinatorial Approach to Testing Deep Neural Network-based Autonomous Driving Systems

The chapter contains a paper published in IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2021.

A Combinatorial Approach to Testing Deep Neural Network-based Autonomous Driving Systems*

Jaganmohan Chandrasekaran[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2]

[1]Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, USA

***Abstract*—**Recent advancements in the field of deep learning have enabled its application in Autonomous Driving Systems (ADS). A Deep Neural Network (DNN) model is often used to perform tasks such as pedestrian detection, object detection, and steering control in ADS. Unfortunately, DNN models could exhibit incorrect or unexpected behavior in real-world scenarios. There is a need to rigorously test these models with real-world driving scenarios so that safety-critical bugs can be detected before their deployment in the real world.

In this paper, we propose a combinatorial approach to testing DNN models. Our approach generates test images by applying a set of combinations of some basic image transformation operations to a seed image. First, we identify a set of valid transformation operations or simply transformations. Next, we design an input parameter model based on the valid transformations and generate a t-way (t=2) combinatorial test set. Each test represents a combination of transformations, and can be used to produce a test image. We execute the test images on a DNN model and distinguish between consistent and inconsistent behavior using a relation. We conducted an experimental evaluation of our approach on three DNN models that are used in the

Udacity challenge. Our results suggest that test images generated by our approach can effectively identify inconsistent behaviors and can significantly increase neuron coverage. To the best of our knowledge, our work is the first effort to use a combinatorial testing approach to generating test images based on image transformations for testing DNNs used in ADS.

Keywords—*Testing DNN models, Combinatorial Testing, Deep Learning Testing, Neural Network Testing, Testing Self- driving cars, Testing autonomous vehicles*

4.1. INTRODUCTION

Recent years have seen significant advancements in the field of deep learning. For traditional software applications, a developer explicitly writes the programming logic based on a specific set of requirements. In contrast, deep learning software applications use a deep neural network (DNN) to derive its decision logic from a training dataset, which typically includes a large number of data instances. Deep learning applications have exhibited an extraordinary ability to discover valuable insights and derive complex decision logic from the training dataset. They have been used to perform tasks, such as image recognition, object detection, and language translation, with a high degree of precision.

Deep learning has been applied in many application domains that are considered to require human intelligence. In particular, deep learning plays a significant role in the operation of autonomous driving systems, where DNN models are used to perform tasks such as obstacle detection, pedestrian detection, steering control, perception and localization, and route planning. However, since DNN models are trained and evaluated using a training dataset, they may suffer from the generalizability problem. For example, an investigation into Uber's accident suggests that their driving software system failed to consider the scenario of jaywalking pedestrians [38]. Tesla's autopilot failed to distinguish between a bright sky and a white trailer crossing an intersection; the

autopilot attempted to drive under the trailer resulting in a loss of life [34]. Accidents reported in [34, 38] suggest a critical need to rigorously test DNN models, especially using tests that imitate the real-world conditions and include corner-case scenarios.

Recent work suggests that synthetic images generated using image transformation techniques can effectively identify the inconsistent behavior of DNN models [36, 44, 46]. Zhang et al. proposed a framework that uses a Generative Adversarial Network (GAN) based unsupervised technique to generate synthetic images that mimic the two extreme weather conditions (snow and rain) [46]. The findings from their study suggest that the DNN models used in the Udacity driving challenge exhibit several inconsistencies when executed with test inputs generated using their approach. Tian et al. demonstrated that testing the DNN model with synthetic images generated with basic image transformations can produce inconsistent behavior [36]. Their results suggest that synthetic images generated by combining different image transformations increase neuron coverage, a measure of proportion of neurons activated in a DNN model.

This paper presents a combinatorial testing-based approach to generating test images to test DNN models. In our approach, we first identify a set of basic image transformations that do not change the ground truth of the image being transformed. That is, in principle, the prediction result for a transformed image produced by such a transformation is the same as the original image. (In practice, the prediction result of the transformed image may be different from that of the original image by a small amount that is less than a certain threshold.) We then use Combinatorial Testing (CT) to generate a t-way test set that covers every t-way combination of these transformations. Each test is a combination of transformations and can be used to create a test image.

To address the test oracle problem, we consider how to identify inconsistent behaviors in two cases. In the first case, the ground truth of a test image remains the same as that of the original image. Thus, we consider that an inconsistent behavior is detected if the prediction result of a test image differs from that of the original image by an amount that is more than a threshold. In the second case, the ground truth of a test image may be different from that of the original image. Thus, the prediction result of a test image may be expected to be very different from that of the original image. In this case, we compare the prediction results of the same test image from different DNNs that perform the same prediction. An inconsistent behavior is detected if the prediction results of a test image from different models do not agree with each other.

Our approach's novelty lies in the fact that we generate test images using CT. The key insight behind CT is that while the behavior of a system could be affected by many factors, individual failures are typically caused by a very small number of factors [21]. We hypothesize that this insight also applies to testing DNNs. That is, inconsistent behaviors of a DNN model could be triggered by a combination of a small number of basic image transformations. In another word, a t-way test set that covers every t-way combination of image transformation can be effective to detect inconsistent behaviors.

We report an experimental evaluation of our approach using three of the top five models, namely Autumn [4], Chauffeur [7], and Rambo [29], from the Udacity self-driving challenge. We generate tests by applying t-way transformations to the seed images selected from the Udacity test dataset. Our results show that t-way tests can identify a number of inconsistent behaviors in these DNN models. For example, out of 121 t-way tests generated for a seed image, 29 tests and 95 tests resulted in an inconsistent behavior for the Autumn model and Chauffeur model, respectively. Our results suggest that a small number of tests (121 tests) can significantly increase the cumulative

neuron coverage compared to its baseline. In some cases, t-way tests covered more than ten times of additional neurons compared to their respective baseline. Overall, the results provide initial support for our hypothesis. The results indicate that t-way tests can help the practitioners to effectively test DNN models in terms of both detecting inconsistent behavior and increasing neuron coverage.

Combinatorial testing is applied to test DNN models, as reported in [9, 23]. However, they follow a white box testing approach by testing the neurons' interactions within each layer in the DNN [23] and the effect of variable strength-based CT tests on interactions between pre-layer and post-layer neurons of the DNN [9]. To the best of our knowledge, we believe the work reported in this paper is the first effort to apply the combinatorial testing approach to generate test images by combining different types of image transformations to test DNN models used in autonomous driving systems.

The remainder of this paper is organized as follows. In Section 4.2, we provide a brief introduction to DNN based software systems and combinatorial testing. In Section 4.3, we present our approach, in terms of the major steps performed in the testing process. In Section 4.4, we report an experimental evaluation, where we first report the design of the evaluation and then discuss the experimental results. Section 4.5 discusses the existing work that is related to ours. Section 4.6 provides concluding remarks and directions for our future work.

4.2. BACKGROUND

4.2.1. DNN BASED SOFTWARE SYSTEMS

Deep learning is a machine learning technique that uses DNN to perform tasks such as classification and regression. In traditional software systems, a developer derives rules from the

requirements and implements the rules in the form of program logic. In contrast, DNN based software systems derive their decision logic from an input dataset; the decision logic is referred to as a trained DNN model. The DNN model takes an input (either image or text depending on the domain) and produces an output in the form of a prediction.

In recent years, DNN models are widely adopted across different domains such as medical imaging, language translation, and autonomous driving systems. They are increasingly deployed in safety-critical fields to perform tasks such as speech recognition, image classification, natural language processing. In particular, autonomous driving systems (ADS) use DNN models to perform tasks such as lane control, object identification, and pedestrian detection. For example, a DNN model used in the autonomous driving system takes an image from the camera as its input and predicts the steering angle.

Based on the application domain, the practitioners use different types of DNN architectures to build a DNN model. Convolutional Neural Network (CNN), a type of neural network architecture, is widely used in the autonomous driving system as they exhibit a higher success rate (better accuracy) in image recognition. Recurrent Neural Network (RNN), a type of neural network architecture that uses temporal information to make predictions, is used in the autonomous driving system to predict steering angles based on a sequence of input data (temporal information). The subject models used in our experiments use a CNN to extract features from the input images that are passed to either an RNN or a fully connected network (FC-network) to predict the steering angle.

4.2.2. COMBINATORIAL TESTING

Combinatorial Testing is a black-box test generation technique. For a given system under test (SUT), combinatorial testing focuses on systematically testing the interactions among the system's different parameters with a smaller number of tests.

Consider a program P with four parameters and each parameter having three values. To test program P, we will require 81 tests (3*3*3*3=81) to test all possible combinations (exhaustive test set). Compared to this, using a t-way combinatorial test set (t=2), it is possible to test all possible interactions between any two parameters (at least once) with nine tests. In general, combinatorial testing approach can significantly reduce the number of tests [10].

ACTS, a combinatorial test generation tool, uses the IPOG algorithm to generate t-way tests. Consider a program P modeled with an input parameter model (IPM) with k parameters. For any t parameters (out of k) of P, IPOG algorithm generates a t-way test set to cover the first t parameters and then it generates additional tests (i.e., extending the test set) to cover the first t+1 parameters in an iterative manner until all the parameters are covered by the test set [43]. ACTS can generate t-way tests of strength t=2 through t=6.

4.3. APPROACH

In this section, we present a combinatorial approach to test DNNs. Figure 4-1 presents an overview of our approach. The proposed approach is applicable for DNNs that take an image as an input and outputs a prediction. The goal of our approach is to generate synthetic images to test the pre-trained DNN model.

In the first step, we identify basic image transformations that can be used to create synthetic test images. Geometric image transformation techniques such as linear transformations and affine transformations can be used to generate synthetic images. Applying a linear transformation to an

image does not change the size or shape of the input image. In contrast, applying the affine transformation, the origins of the transformed image and the original image do not necessarily match with each other. In other words, applying the affine transformation shall result in a change of orientation and size of the original image. We represent a transformation in a two-tuple form *(transformation name, transformation value)*. For example, (Brightness, 10) increases an image's brightness by a value of 10.

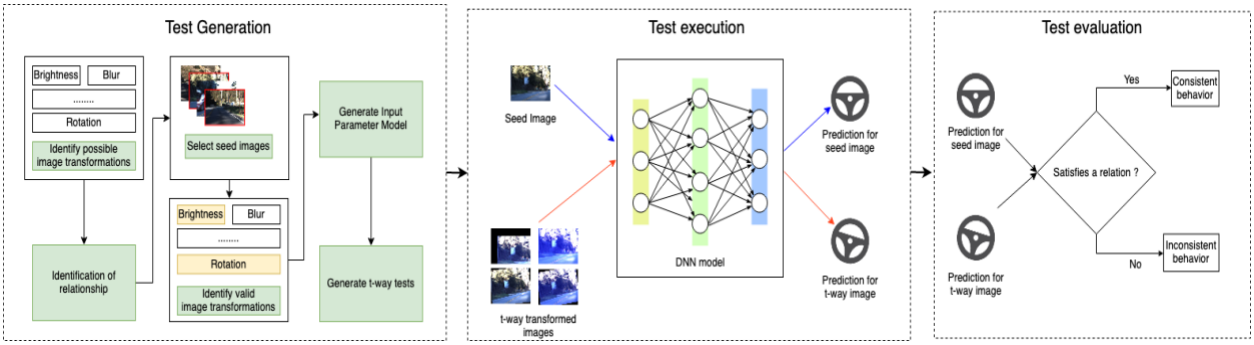


FIGURE 4-1 APPROACH OVERVIEW

In the second step, given the nature of the domain, the test input space for a DNN model can be too large (nearly infinite). Hence, we apply equivalence partitioning, and randomly select a seed image from each partition. For example, a test dataset (used in Autonomous Driving Systems domain) contains image frames recorded all around the year. In this case, we partition the test dataset based on the weather conditions such as sunny, rainy, fog, snow, overcast, and normal weather. We then randomly select an image (seed image) from each group.

Our approach is aimed to generate valid test images (synthetic images). In the first step, we identified a set of transformations that could be applied to generate synthetic images. However, every transformation might not be uniformly applicable to seed images. In other words, given the type and nature of the seed image, applying certain transformations (identified from Step 1) might generate a synthetic image that is either unrealistic or invalid. For example, consider two seed

images: image #1 captured during the middle of the day (brighter, sunny day) and image #2 captured around the time of sunset during winter. We generate a synthetic image by applying a transformation -- decrease the brightness by 80% to both seed images. The resulting synthetic image for image #1 could be valid in terms that the image is viewable to human eyes. In contrast, the synthetic image for image # 2 might be invalid, since it could be a completely dark image. As this example illustrates, some image transformations when applied to a seed image might generate invalid test inputs. Note that, in this case, we consider image #2 as invalid. However, in real world, it is still important to test such scenarios using other approaches that deal with images that are completely dark.

To alleviate this problem, in the third step, for each seed image, we identify a set of valid transformations (a subset of all possible transformations). We determine the validity by comparing the prediction results of the original image (Po) and the transformed image (Ps). For DNN models that outputs a continuous value (for example, a steering angle), the transformed image's prediction result may be different, within a degree of tolerance, from that of the original image. Therefore, a transformation is considered valid if the difference between Po and Ps is less than a certain threshold $|Po - Ps| \leq$ threshold.

Consider the following example. We are testing a pre-trained DNN model that predicts the steering angle. Contrast and Rotation are two possible image transformations, with five different values per transformation: (Contrast,1), (Contrast,2), (Contrast, 3), (Contrast, 4), (Contrast, 5), and (Rotation, 2°), (Rotation, 4°), (Rotation, 6°), (Rotation, 8°), (Rotation, 10°). Thus, there exists a total of ten possible transformations. We apply these transformations to the seed image and generate ten synthetic images. Then, the predicted value of a synthetic image is compared with the predicted value of the original image. Three transformations – (Contrast, 5) and (Rotation, 8°),

(Rotation, 10°) exceed the threshold. The remaining seven transformations are identified as valid transformations for the seed image. It is often the case that different seed images can have different sets of valid transformations. The motivation behind this step is to generate valid tests, thus minimizing false positives.

In the fourth step, we generate t-way tests. For each seed image, we design an Input Parameter Model (IPM), where each transformation is identified as a parameter, and the transformation values that make a transformation valid are identified as parameter values. In our earlier example, Contrast: {1,2,3,4} and Rotation: {2°, 4°, 6°} are identified as parameters and values. In the final step, based on the IPM, we generate abstract t-way test set. Then, we derive concrete tests (synthetic images) by applying t-way image transformations to the seed images using the OpenCV framework [49]. The synthetic images are used to test the DNN models.

One challenge in testing ML models is lack of a test oracle. In practice, data labeling is considered to be an expensive and challenging task. Thus, a tester might not be able to determine the ground truth of a synthetic image. In this case, the practitioners can compare the prediction value across different model implementations and identify the inconsistent behavior. Doing so can help practitioners assess a model's performance in the absence of ground truth. In our approach, we evaluate the t-way test results in the following two cases:

Case 1: The original seed image and t-way synthetic image share the same ground-truth value. In this case, for each model, if a test fails to satisfy the relation: $|Po - Ps| \leq$ threshold, the test is considered to exhibit inconsistent behavior.

Case 2: The original seed image and t-way synthetic image do not share the ground-truth value, i.e., they might have a different ground-truth value. In this case, we evaluate a test by comparing its prediction results across multiple models (Pm). It can be challenging to derive the

ground truth for each test (synthetic image). Therefore, we define an inconsistent behavior as follows: A test exhibits an inconsistent behavior if the maximum difference in prediction change across multiple models exceeds a threshold value i.e., if a test fails to satisfy the following relation

$$|\max(\mathrm{Pm}) - \min(\mathrm{Pm})| \leq \text{threshold} \qquad (1)$$

4.4. EXPERIMENTS

In this section, we present an experimental evaluation of our approach. The source code, data and/or artifacts have been made available at [31, 35].

4.4.1. RESEARCH QUESTIONS

Our experiments are designed to answer the following two research questions:

- Can our combinatorial testing-based approach successfully identify inconsistencies among DNN model implementations?
- How does the combinatorial testing-based approach impact the neuron coverage?

4.4.2. MODELS

We use open-source DNN models from the Udacity self-driving car challenge. Teams participating in the Udacity self-driving car challenge developed DNN models that predict the steering angle (output) based on an image frame (input). Submitted models were evaluated with the Udacity test dataset [30] and ranked based on their prediction accuracy (performance). Models from the Udacity self-driving car challenge are among the widely used subject models to evaluate test generation techniques for testing autonomous vehicle software systems [36][44][46][15].

Among the top five ranking models from the challenge that are publicly available at [40], we select three models, namely Chauffeur [7], Rambo [29], and Autumn [4] as our subject models. We did not use the other two models, namely, komanda [40] and rwightman [40]. For komanda,

the pre-trained model weight file is not accessible [20]. For rwightman, the publicly available script failed to execute [30].

- The Autumn model consists of three 5x5 convolution layers with stride 2, followed by two 3x3 convolution layers and five fully connected layers with a dropout [4]. The Autumn model is implemented using Tensorflow(v0.11) and Keras(v1.1.0) [1, 18].
- The Chauffeur model uses a Convolutional Neural Network (CNN) to extract features from the input image and use a Long Short-Term Memory (LSTM) network, a type of Recurrent Neural Network (RNN), to predict the steering angles. Chauffeur model is implemented using Tensorflow (v1.12) and Keras (v1.2.2) [1, 18].
- The Rambo models consist of three CNNs to extract features, and their output is merged in the final layer to predict the steering angle. The Rambo model is implemented using Tensorflow (v1.12) and Theano (v0.9) [1, 33].

For each subject model, the sequence of image frames that has been processed before the current frame impacts the prediction of the current frame. In Autumn, the prediction is based on five consecutive frames (input + four previous frames). Chauffeur's prediction is determined by 100 consecutive frames (input + ninety-nine previous frames). Rambo considers three consecutive frames to make the prediction (input + two previous frames).

We present the model details in Table 4-1. The first and second column list the model name, and its network architecture. The third column presents the Root Mean Square Error (RMSE) value. RMSE is one of the widely used metric to measure the prediction errors of a machine learning model that outputs a continuous value. A lower RMSE value indicates better performance (prediction). All submitted models in the Udacity challenge were evaluated and

ranked per RMSE value. In the last column, we present information about the number of sequence images that influence the current frame's prediction.

TABLE 4-1 MODEL INFORMATION

MODEL NAME	ARCHITECTURE INFORMATION	RMSE		PREDICTION LOGIC
		REPORTED RMSE	OUR RMSE	
Autumn	CNN	0.04	0.04	Previous 4 frames + current frame
Chauffeur	CNN + RNN	0.06	0.06	Previous 99 frames + current frame
Rambo	CNN	0.06	0.06	Previous 2 frames + current frame

4.4.3. SEED IMAGES

We select the seed images from the Udacity test dataset. The test dataset consists of 5614 test images and their respective steering angles [39]. The steering angle is in the range -25° to +25° and normalized to ±1° [36]. An image with a positive steering value indicates the vehicle is turning right. A negative steering value indicates turning left, while a steering angle of 0° or closer to 0° indicates the vehicle is traveling in a straight direction (i.e., no turns).

The steering angle is in the range of -1 to +1. Based on the steering angle, we divide the test images into different groups with an interval of 0.1 per group. We have a total of 20 groups starting from ($-1.0 <$ steering angle ≤ -0.9) through ($0.9 <$ steering angle ≤ 1.0). We refer to these groups by Group 1 through Group 20, respectively.

The test dataset does not contain images in the range (-1.0, -0.9). Thus, there is no representative image from Group 1 in our experiments. For the remaining nineteen groups, we

randomly select one image from each group as our seed image. In total, we have nineteen seed images in our experiments.

4.4.4. TEST ORACLE

In the autonomous driving domain, it is hard to determine an exact steering angle for transformed images. Zhang et al. used a method to identify the DNN model's consistent behavior, and it is defined as follows: Given a transformed image as input, if the DNN model predicts (steering angle) within a certain error bound, the model is considered to exhibit a consistent behavior [46]. Similar to their work, we use a relation to identify a model's inconsistent behavior in two cases.

In the first case, we assume the t-way synthetic image and the original image shares the ground truth. Thus, a t-way synthetic image that violates the relation $|Po - Ps| \leq$ threshold exhibits an inconsistent behavior. Po denotes the steering angle of the original image and Ps denotes the steering angle of the transformed image.

In the second case, we assume the synthetic image and the original image does not share the ground truth. In this case, we compare the prediction results of the same synthetic image from three DNNs that perform the same prediction. A t-way synthetic image exhibits an inconsistent behavior if it violates the relation (1).

The threshold value is a configurable parameter, and we use the following three threshold values: 0.1, 0.2 and 0.3 in our experiments.

4.4.5. METRICS

We measure our approach's effectiveness by computing the number of inconsistent behaviors identified by a t-way test set. The more inconsistent behaviors the t-way test detects, the more effective the t-way test is considered.

We also use neuron coverage to measure the effectiveness of our approach. The notion of neuron coverage is defined as the ratio of unique neurons that is activated for a given input to the total number of neurons in a DNN [27]. A neuron is considered activated if its output is greater than a certain threshold (defined by the user). Tian et al. used neuron coverage in their experiments and made their artifacts publicly accessible [3, 36]. We use their neuron coverage framework and threshold (0.2) in our experiments. To measure the cumulative neuron coverage, we first load the seed image to the DNN and measure its neuron coverage. This coverage information is used as the baseline in our experiment. Then, we execute the t-way images and calculate cumulative coverage relative to the baseline.

4.4.6. TEST GENERATION

We begin the test generation step by identifying the possible image transformations applicable to the Udacity test dataset. Tian et al. applied a set of seven different types of simple image transformations to the Udacity test dataset and studied their impact on neuron coverage [36]. We use these seven image transformations. Table 4-2 presents the list of transformations and their values used in our experiments. Overall, we have seventy image transformations (7 different types of transformations * 10 values per transformation).

Recall that, as discussed in section 4-3, every possible transformation might not be uniformly applicable to all the seed images. Therefore, in the next step, we identify the set of valid transformations for each seed image.

4.4.6.1. IDENTIFICATION OF VALID TRANSFORMATIONS

We apply the seven types of transformations with ten different values per transformation and generate 70 synthetic images per seed image. Next, the seed image is loaded to three subject models, and their respective predicted steering angle is recorded (Po).

Then, for each model, the synthetic images are loaded as input. Their predicted steering angle (Ps) is compared with the steering angle of the original seed image (Po). A transformation is considered to be valid if $|Po - Ps| \leq 0.1$. At the end of this step, we identify the set of valid transformations per seed image.

TABLE 4-2 TRANSFORMATIONS AND VALUES

TRANSFORMATIONS		**VALUES**
Blur	Averaging	3x3, 4x4, 5x5, 6x6
	Gaussian	3x3, 5x5, 7x7
	Median	3, 5
	Bilateral	(9, 75, 75)
Brightness		10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Contrast		1.2, 1.4, 1.6, 1.8, 2.0, 2.2., 2.4, 2.6, 2.8, 3.0
Rotation		3, 6, 9, 12, 15, 18, 21, 24, 27, 30
Scale		1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0
Shear (Horizontal)		0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
Translation		(10,10), (20,20), (30,30), (40,40), (50,50), (60,60), (70,70), (80,80), (90,90), (100,100)

4.4.6.2. GENERATION OF T-WAY TESTS

To create t-way tests, first, we create an input parameter model (IPM). The seven transformations from Table II are identified as parameters. Based on the valid transformations, we identify the set of possible values for each parameter. Next, based on the IPM, we generate abstract t-way tests using the ACTS tool [2]. In our experiments, we generate a 2-way combinatorial test set. Each test represents a combination of transformations that could be applied to the seed image. In the final step, we convert the abstract t-way tests to concrete tests, i.e., generate synthetic images based on t-way tests. Recall that each abstract test represents a combination of transformations. When we generate synthetic images, we apply image transformations in the following order – *Blur, Brightness, Contrast, Rotation, Scale, Shear and Translation* using the OpenCV framework [49]. (We tried to apply the transformations in different orders, and found that the order has minimal impact on the prediction outcomes.) We execute the subject DNN models with t-way concrete tests (i.e., synthetic images) and compare their output with the original image's predicted steering angle.

4.4.7. EXAMPLE

We illustrate our approach with an example. For the seed image from group 2 (*1479425660620933516.jpg*), the three models, namely Chauffeur, Rambo, and Autumn, predict a steering angle -0.760681748390198, -0.62006545, and -0.83253384, respectively (Po).

In Step 1, we generate 70 synthetic images for *1479425660620933516.jpg* based on the transformations listed in Table 4-2. Then, we execute the 70 synthetic images on three models and compare their predicted steering values (Ps) with their respective steering angle prediction of the original image (Po).

In the case of the Chauffeur model, a transformation (synthetic image) is considered valid if $|Po - Ps| \leq 0.1$; 48 out of 70 transformations satisfy the criteria and thus considered valid

transformations for the chauffeur model. For the Rambo model, a transformation (synthetic image) is considered valid if $|Po - Ps| \leq 0.1$; 57 out of 70 transformations satisfy the criteria. Likewise, for the Autumn model, a transformation is considered valid if the absolute value of $|Po - Ps| \leq 0.1$, and 33 transformations satisfy the criteria. Among the 70 transformations, 28 transformations are valid across all three models, and hence these 28 transformations are used to generate t-way tests.

In step 2, using the ACTS tool, we create the input parameter model with valid parameters and values identified from the previous step. Then, we generate 121 abstract tests from a 2-way test set. Next, we use the Open-CV framework [49] to generate concrete tests (2-way synthetic images). Finally, we test the subject models using concrete tests.

4.4.8. Results and Discussion

First, we present the results of synthetic images generated using the transformations and values from Table 4-2. These results are used to identify valid transformations that are later used in the generation of t-way tests. Next, we present the inconsistent behavior detection results of the t-way tests. Finally, we discuss the neuron coverage achieved by the t-way tests.

4.4.8.1. Identification of Valid Transformations

Figure 4-2 presents the details of valid transformations identified for each group. The x-axis presents the group details, and the y-axis presents the number of possible transformations. In our experiments, we have 70 possible transformations (7 transformations * 10 values per transformations). Due to limited space, we present the number of valid transformations per group. Our results suggest that *Group 10* has the maximum number of valid transformations, i.e., 50 out of 70 transformations are valid. *Group 20* has the minimum number of valid transformations, i.e., only 16 out of 70 transformations are valid. We observe that all transformations *Blur* are valid

across 18 groups. On the contrary, five transformations, namely Rotation_24, Rotation_27, Shear_0.4, Shear_0.5, and Shear_0.6 were invalid across all groups as they failed to meet our criterion (Po-Ps<=0.1 for all three models).

4.4.8.2. INCONSISTENT BEHAVIOR DETECTION RESULTS OF T-WAY TESTS

In this section, we present the results of t-way synthetic images. We generate 2-way tests based on the set of valid transformations identified from the previous step for each group. Each test represents a combination of transformations that could be used to generate synthetic image. Then, we execute the 2-way tests in three subject models to identify the number of consistent and inconsistent behaviors among three DNN models.

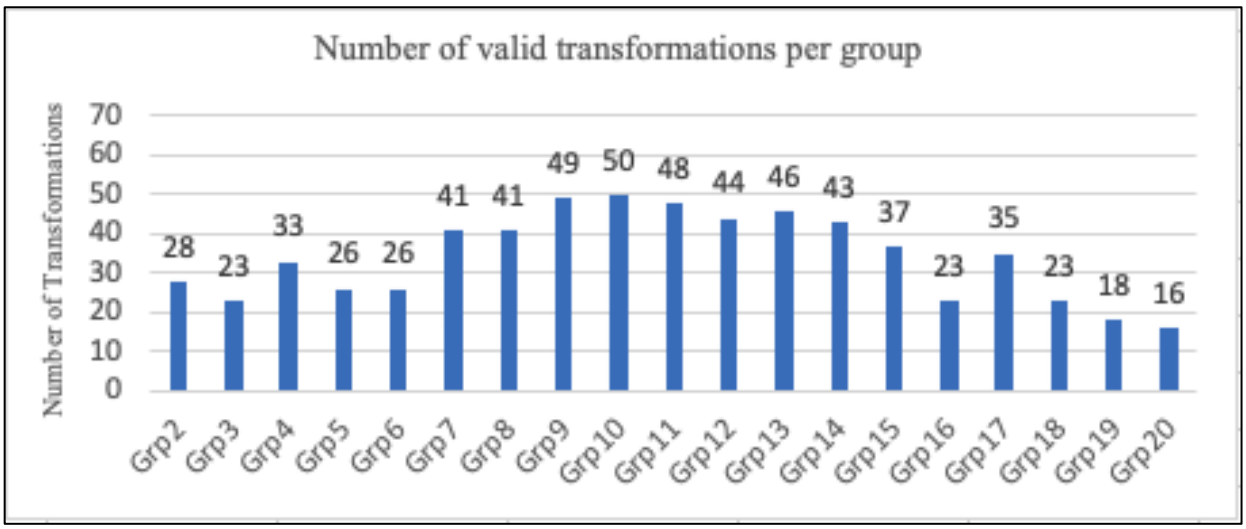


FIGURE 4-2 NUMBER OF VALID TRANSFORMATIONS FOR EACH GROUP

Results for Case 1: In this case, we assume that the original seed image and t-way synthetic image share the ground-truth value. Figures 4-3, 4-4, and 4-5 present the t-way test results for threshold values of 0.1, 0.2, and 0.3, respectively. Recall that we generate t-way tests per group, and the total number of t-way tests varies among the groups. Therefore, we present our results as a percentage of t-way tests that exhibit consistent behavior. The x-axis represents the group

number. The y-axis represents the percentage of t-way tests that exhibit consistent behavior. The last column in Table 4-3 presents the total number of t-way tests generated for each group.

For a threshold value of 0.1, our results indicate that *Rambo* is less prone to inconsistent behavior among the three subject models. As Figure 4-3 suggests, for eight groups (3, 7, 9, 12, 13, 15, 16, and 20), t-way tests executed with the *Rambo* model do not display any inconsistent behavior (all tests result in a passing state). In addition to this, in groups 5, 6, 14, and 18, more than 90% of the t-way tests executed with the *Rambo* model result in a consistent behavior. Apart from Group 10, the *Rambo* model exhibits a better prediction performance than the other two models. In the case of *Chauffeur*, more than 50% of t-way tests generated for seven groups (2, 7, 8, 9, 12, 14, 18) results in an inconsistent behavior; the lowest being Group 12 with a meager 16% of tests resulting in a consistent behavior. On the contrary, for the same group (Group 12), 96% of the t-way tests result in a consistent behavior for the *Rambo model*. Our results suggest that the *Autumn* model exhibits a mixed performance. In a few cases (Groups 6 and 10), more than 90% of t-way tests result in a consistent behavior state. On the contrary, for six groups (Group 2, 3, 8, 17, 18, 20), the *Autumn* model produces an inconsistent behavior for more than 50% of the t-way tests.

Figures 4-4 and 4-5 suggest an increase in the threshold value results in better performance, i.e., a higher number of consistent behaviors across three models. In the case of Rambo, with a threshold of 0.2, in most cases, all t-way tests generated exhibit a consistent behavior (16 out of 19 groups). However, for the rest of the two models, we observe that more than 25% of t-way tests still result in inconsistent behavior for some groups. For example, group 14, 17, 18, and 20 for Autumn, group 2, 8, 9, 12, 16, and 18 for Chauffeur (threshold 0.2). We observe a similar pattern

for threshold 0.3. Overall, the results suggest that the *Rambo* model exhibits better performance than the other two models.

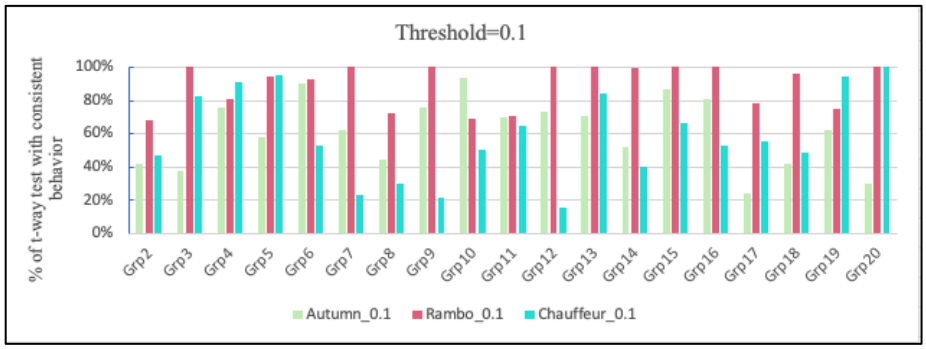


FIGURE 4-3 T-WAY RESULTS FOR THRESHOLD 0.1 (CASE #1)

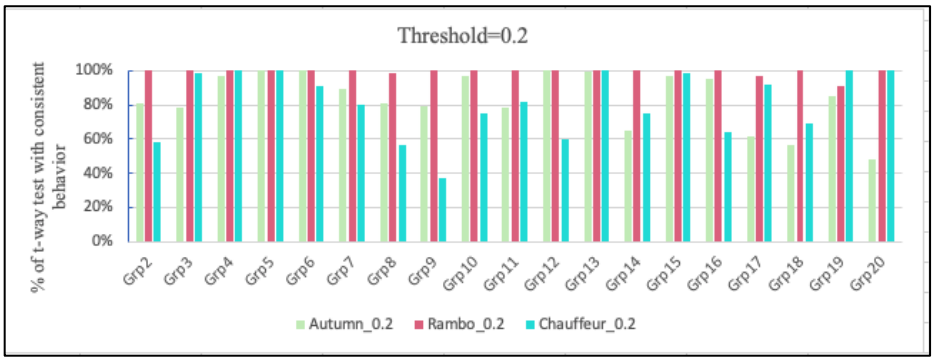


FIGURE 4-4 T-WAY RESULTS FOR THRESHOLD 0.2 (CASE #1)

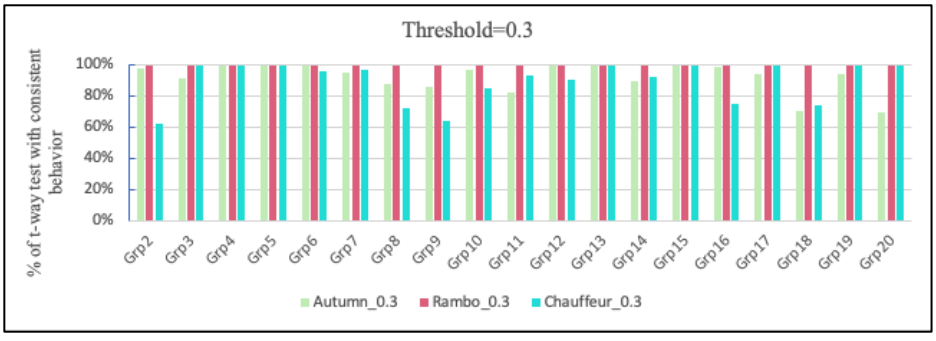


FIGURE 4-5 T-WAY RESULTS FOR THRESHOLD 0.3 (CASE #1)

Results for Case 2: Recall that in this case, the original and t-way synthetic images might have a different ground-truth value. We evaluate the t-way test results with three threshold values: 0.1 (2.5°), 0.2(5°), and 0.3 (7.5°). Table 4-3 presents the results. The first column lists the group number. The next three columns present the number of t-way tests exhibiting inconsistent behavior for thresholds of 0.1, 0.2, and 0.3, respectively. The last column presents the total number of t-way tests for each group.

The result suggests that t-way tests can detect a significant number of inconsistent behaviors across different thresholds. For a threshold of 0.1, in 18 (out of 19) groups, 50% or more tests result in inconsistent behavior. In 12 (out of 19) groups more than 90% of tests results in inconsistent behavior.

Our results indicate that an increase in the threshold value results in a decrease in the number of inconsistent behaviors. This is as expected. With a threshold of 0.3 (7.5°), for four groups (Group 5, Group 12, Group 13, and Group 15), less than 3% of tests resulted in inconsistent behavior. This indicates that a further increase in threshold might result in a large number of false negatives. Therefore, we did not consider a threshold value that is larger than 0.3.

Overall, the results suggest that t-way test set are effective in identifying model inconsistencies. We acknowledge that both Case 1 and Case 2 have limitations. In Case 1, in some scenarios, determining the ground truth for a synthetic image generated from a t-way test set can be a challenging task (lack of test oracle). In Case 2, given the nature of differential testing, a model inconsistency can be detected only if (1) there exist at least two or more models implementing the same functionality, and (2) at least one model producing a different result. A practitioner shall choose between Case 1 and Case 2 based on their domain knowledge.

TABLE 4-3 NUMBER OF INCONSISTENT BEHAVIOR IDENTIFIED BY T-WAY TESTS (CASE #2)

GROUP NUMBER	# OF INCONSISTENT BEHAVIORS PER THRESHOLD			TOTAL # OF TESTS
	0.1	0.2	0.3	
2	119	102	69	121
3	107	89	53	110
4	100	45	13	110
5	86	29	0	121
6	102	65	27	102
7	120	86	23	122
8	114	95	55	121
9	109	95	75	121
10	96	54	27	121
11	66	39	27	122
12	91	15	3	126
13	95	30	4	121
14	102	85	36	122
15	44	7	0	121
16	121	118	85	121
17	121	110	78	121
18	54	47	35	55
19	55	52	34	55
20	30	23	18	33

4.4.8.3. T-WAY TESTS AND THEIR IMPACT ON NEURON COVERAGE

In this section, we present the neuron coverage achieved by t-way tests for the Rambo model. The Rambo model consists of 3 CNN sub-models referred to as S1, S2, and S3, and they consist of a total of 1625, 3801, and 13473 neurons, respectively [36]. Overall, the Rambo model consists of 18899 neurons.

Table 4-4 presents the neuron coverage for the seed images (baseline). The results indicate that most of the seed images (17 out of 19) cover approximately 10% of the total neurons, while Group 10 and Group 11 cover 73.29% and 71.78% of the total neurons, respectively.

Next, we present the neuron coverage achieved by the t-way tests in Figures 4-6, 4-7, 4-8, and 4-9. The x-axis represents the group number. The y-axis represents the percentage of additional neurons covered by the t-way tests compared to their respective baseline. Results suggest that t-way tests result in a significant increase in neuron coverage. In the case of S1, we notice a moderate increase in the additional number of neurons covered compared to the baseline. The result presented in Table 4-4 indicates, amongst the nineteen groups, the seed image representing Group 17 achieves the least coverage for S1 with 460 neurons. The t-way tests generated for group 17 cover an additional 25% of neurons (113 neurons) compared to its baseline. Similarly, in sub-model S2, across groups, we notice a substantial number of additional neurons covered by the t-way tests; seven groups covering more than 50% of additional neurons compared to their respective baseline.

We observe that t-way tests achieve a significant increase in neuron coverage for sub-model S3. Out of 19 groups, t-way tests generated for eleven groups achieve more than one hundred percent increase in cumulative neuron coverage; six groups (Group 2, 7, 8, 9 12, 13) achieve more than ten times increase in cumulative neuron coverage. On the contrary, t-way tests

for two groups - Group 10 and Group 11 cover a significantly lesser number of additional neurons. This can be explained as follows: for sub-model S3, the seed images representing Group 10 and 11 cover 95.91% and 94.01% neurons. Hence, their respective t-way tests result in a marginal increase in neuron coverage.

TABLE 4-4 NEURON COVERAGE OF SEED IMAGES (RAMBO)

GROUP NUMBER	**NUMBER OF COVERED NEURONS**			
	S1	**S2**	**S3**	**TOTAL**
2	500	449	802	1751
3	501	452	1113	2066
4	497	416	722	1635
5	501	428	827	1756
6	496	445	718	1659
7	492	433	1153	2078
8	485	461	778	1724
9	483	438	795	1716
10	468	461	12923	13852
11	475	424	12667	13566
12	463	442	960	1865
13	465	422	808	1695
14	471	430	904	1805
15	467	459	806	1732
16	466	433	1224	2123
17	460	466	822	1748
18	480	456	1176	2112
19	486	422	3801	2118
20	469	447	1189	2105

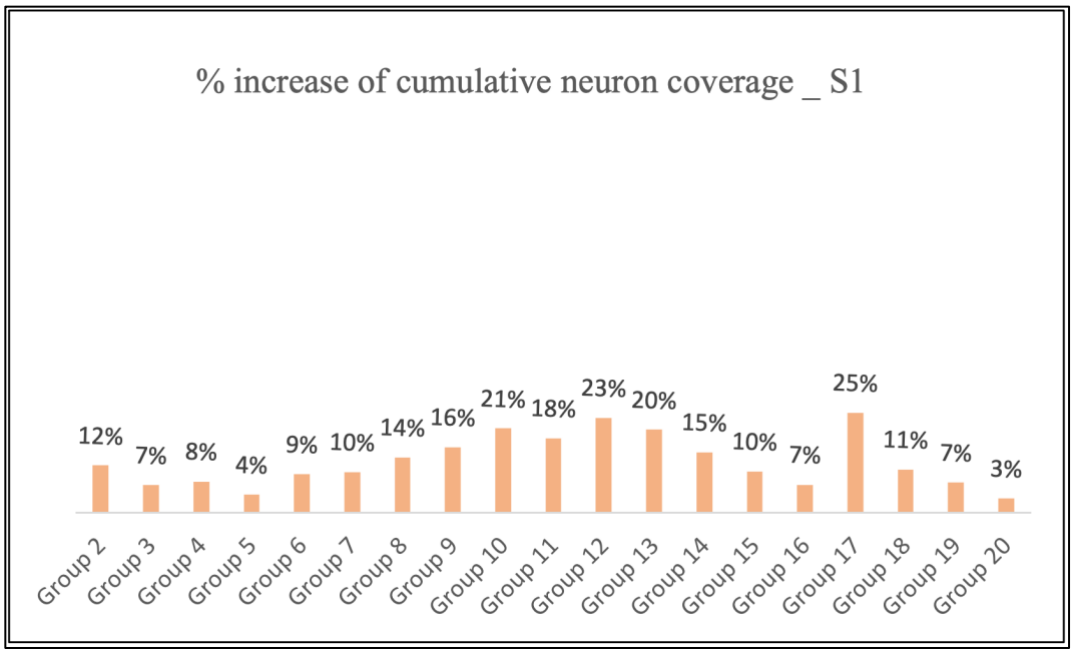


FIGURE 4-6 S1

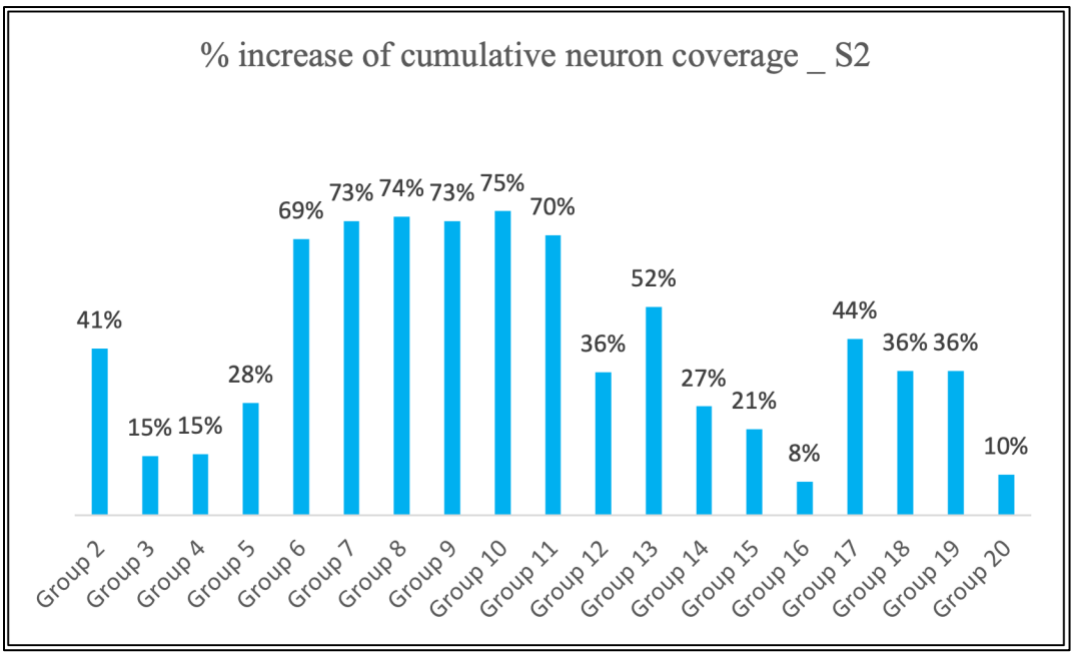


FIGURE 4-7 S2

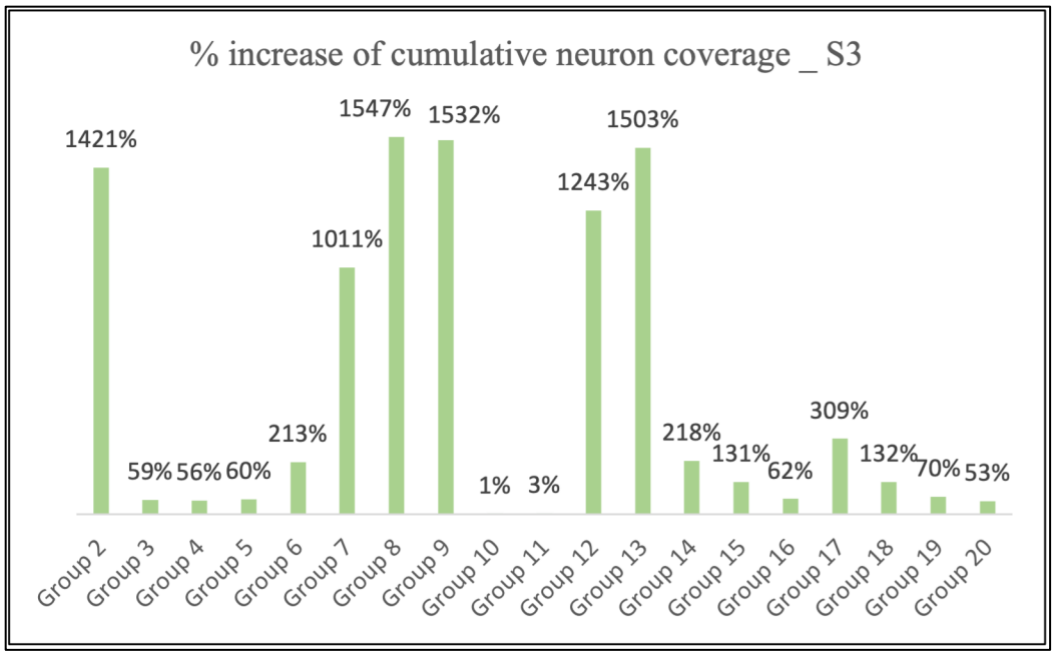


FIGURE 4-8 S3

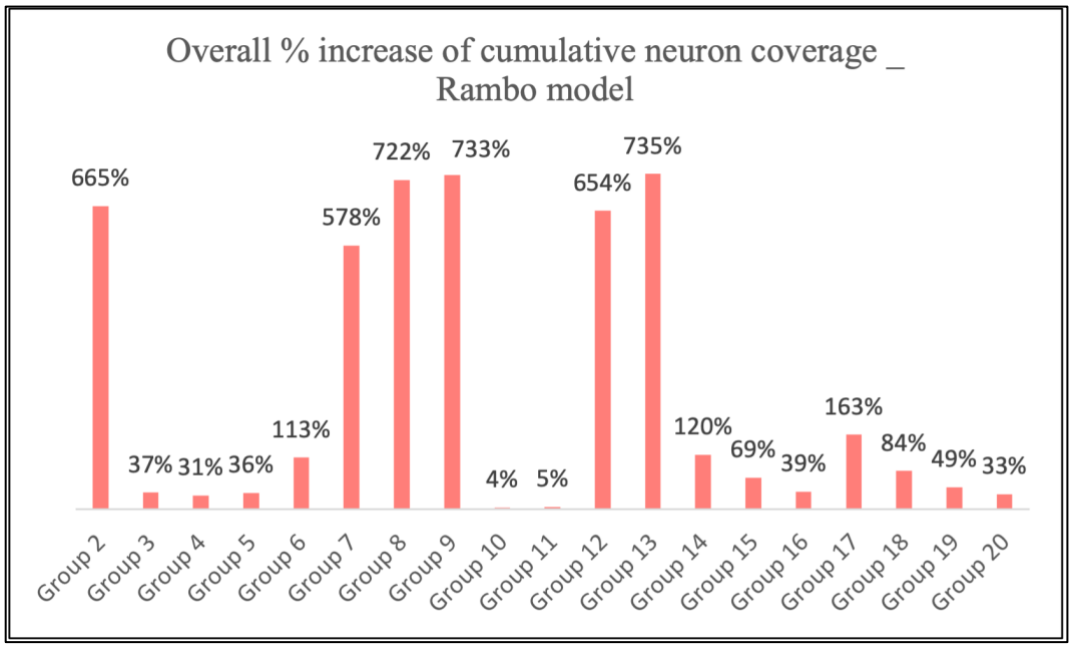


FIGURE 4- 9 RAMBO MODEL

Overall, the result suggests that t-way tests increase the neuron coverage significantly. In some cases, the results suggest a smaller number of t-way tests (120 tests) can cover more than ten times of additional set of neurons.

We acknowledge that the neuron coverage results are unavailable for the remaining two models due to time limitations. On executing the Chauffeur model with a batch of 100 images, the current version of the framework, on average takes 18 minutes to measure the neuron coverage. So, t-way tests for a group (with 121 tests) takes around [(121*18)/60 = 36 hours]. It will take weeks to complete the coverage measurement for all nineteen groups. We plan to study the impact of t-way tests on neuron coverage of Autumn and Chauffer model as a part of future work. Also, we plan to investigate the correlation between neuron coverage and fault detection as a part of future work.

4.4.9. THREATS TO VALIDITY

Threats to external validity occur when the results from our experiments could not be generalized to other subjects. The DNN models used in our study have been used in other studies [36, 44, 46, 15]. All three DNN models used in our experiments have different architectures, thus alleviating the risk of lack of DNN architectures (representativeness) used in our study.

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. To mitigate the risk of human errors, we tried to automate as many tasks as possible, from generating synthetic images to executing the tests. Also, we have manually checked some of the results whenever any inconsistent or surprising results occur. For example, out of 121 tests generated for a seed image (group 8), 72.73% percentage of tests results in a consistent behavior for Rambo model. In contrast, the other two models had less than 50% of the

tests resulting in a consistent behavior. In such scenario, we manually verified the results by analyzing the log file.

4.5. Related Work

We first discuss the existing work related to testing DNN based software systems. Traditional testing techniques such as coverage-guided testing [26, 36, 42], concolic testing [32], mutation testing [24], differential testing [27], combinatorial testing [9, 23] have been applied to test DNN models. We focus on the existing work reported in applying combinatorial testing (CT) to test machine learning systems as they are most relevant to our work.

Ma et al. proposed DeepCT, a combinatorial testing coverage guided test generation technique to test the robustness of the DNN model [23]. DeepCT follows a white box testing approach by testing the interactions of the neurons within each layer in the DNN. Similarly, Chen et al. apply variable strength combinatorial testing to test DNN models. They propose three different methods to construct variable strength-based CT tests and study their effect on interactions between pre-layer and post-layer neurons [9]. In contrast, we apply CT as a black-box approach to generate test images and detect potential predictions errors of DNN models used in autonomous vehicle software systems.

Li et al. proposed an ontology-based test generation framework for testing autonomous driving systems [22]. In Step 1, they construct an ontology based on the autonomous driving domain. In Step 2, they convert an ontology to a combinatorial test input model using conversion algorithms. Next, based on the test input model, they generate abstract tests that are used to create concrete tests. In Step 3, they execute the concrete tests and evaluate their results. Gladisch et al. proposed a combinatorial testing approach to generate a test dataset for testing perception functions [14]. They use SCODE [12] to convert a domain model to an input test model for PICT, a pair-

wise test generation tool [17]. Using PICT, they generate abstract test cases that are later converted to concrete tests (test images).

Similar to [14, 22], we also develop an input parameter model (IPM), generate abstract t-way tests (based on the IPM), generate, execute and evaluate the concrete tests. Our work differs in the following way: Li et al. develop an input test model based on the road parameters such as slop, surface, and lane type. Gladisch et al. generate input test model based on the traffic scenarios such as daytime, sky conditions, rain, reflection on road etc. In contrast, we develop an IPM based on the seven image transformations techniques namely blur, brightness, contrast, rotation, scaling, shearing and translation.

Next, we discuss the existing literature on testing autonomous vehicle software systems. A significant amount of work has been reported on testing autonomous vehicle software systems [11, 13, 15, 19, 25, 27, 36, 37, 46, 47, 48]. Pei et al. proposed a technique to generate synthetic test inputs using a joint optimization problem to test DNN models used in autonomous vehicle systems [27]. Tian et al. proposed an approach to generate test inputs (synthetic images) by simple image transformations [36]. Yan et al. presented an approach that generates tests by Adaptive Random Testing (ART) technique and uses an Adaptive Random Testing for Deep Learning Systems (ARTDL) algorithm that selects test input using a distance metric known as Feature-based Euclidean Distance (FED) to test the model under test [44]. Zhang et al. proposed an unsupervised image-to-image transformations framework based on Generative Adversarial Network (GAN) that generates synthesized test inputs that mimics two weather conditions, namely snow, and rain, to test the DNN model [46]. Haq et al. presented an empirical comparison of offline testing (testing the DNN model as an individual component) and online testing (testing the DNN model as a part of a software system) of DNN models used in autonomous driving systems [15].

Similar to our work, existing work reported in [15, 27, 36, 44, 46] have used the Udacity driving challenge-2 datasets to evaluate their respective approaches. Also, our work is similar to [36, 44, 46], in terms of generating test inputs by image transformations and testing and evaluating the DNN models using metamorphic relations. However, our work differs in the following way. Tian et al. [36] primarily study the impact of synthesized images (generated by combining different transformations) on the neuron coverage. In contrast, our work focusses on evaluating the impact of synthesized images on the model's prediction. The work presented in [15] compared the offline and online testing of DNN systems. It investigated the possibility of testing DNN's by replacing the original dataset with simulator-generated datasets. In contrast, our work explores the possibility of generating test inputs using a combinatorial testing approach to detect prediction errors in DNN models. Zhang et al. use an unsupervised network that uses GAN to generate synthesized test inputs that mimic different weather conditions [46]. Compared to [44, 46], our work is focused on generating tests using a combinatorial testing approach, i.e., generating synthesized images by combining different images transformations. To the best of our knowledge, ours is the first work that applies combinatorial testing techniques to generate t-way synthetic images for testing DNN models used in autonomous driving software systems. We also note that there is a significant number of existing studies in literature, and we refer the reader to [45] for a comprehensive report on existing work on testing machine learning systems.

4.6. Conclusion and Future work

In this paper, we present a combinatorial testing-based approach to systematically generate test images to test DNN models used in the autonomous driving systems. We begin our approach, by applying basic image transformations on the seed image (original) and identifying a set of transformations that do not change the ground truth of the image being transformed as valid

transformations. Then, based on the valid transformations, we develop the IPM and generate t-way tests each of which is applied to the seed image to generate an synthetic image. We identify inconsistent behaviors of DNN models in two scenarios: (1) the original and synthetic image share the ground truth and (2) the original and synthetic image does not share the ground truth.

We performed an experimental evaluation of our approach with three publicly available pre-trained DNN models and datasets from the Udacity self-driving challenge. Our results indicate, for scenario 1, *Rambo* model exhibits a better performance, i.e., less prone to inconsistent behavior, compared to the other two models. For scenario 2, synthetic images generated by combining a set of image transformations (t-way tests) can successfully identify inconsistent behavior among models. With a threshold of 0.1, more than 90% of test cases from 12 groups result in an inconsistent behavior.

Result suggests t-way tests significantly increases the neuron coverage for the *Rambo* model. Out of the 19 groups, synthetic images generated for 17 groups, result in a moderate to significant increase in cumulative neuron coverage; nine groups (Group 2, 6, 7, 8, 9, 12, 13, 14, 17) achieves more than one hundred percent increase in cumulative neuron coverage. Given the time-intensive nature of the measurement process, we are unable to measure the neuron coverage for the remaining two models. We plan to complete the measurement as a part of future work.

This is part of our larger effort in applying combinatorial testing to test DNN based systems. We plan to include additional weather-based transformations such as rain, fog, smog, and shadows to generate test images. We hope to leverage the insights gained from this study to refine our input parameter model, develop realistic and meaningful constraints and thus generating more effective t-way tests to test DNN based systems. Also, we plan to extend this work by investigating

how the combinatorial testing-based approach can be adopted in testing different versions of the DNN models in regression testing.

4.7. ACKNOWLEDGMENT

This work is supported by research grant (70NANB18H207) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

4.8. REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16) (pp. 265-283).

[2] Advanced Combinatorial Testing System (ACTS) | CSRC, https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software/downloadable-tools, Accessed: 2021-02-28

[3] ARiSe-Lab/deepTest: A systematic testing tool: https://github.com/ARiSE-Lab/deepTest, Accessed: 2020-10-12

[4] "Autumn-model:" https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/autumn, Accessed: 2020-09-12

[5] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2014). The oracle problem in software testing: A survey. IEEE transactions on software engineering, 41(5), 507-525.

[6] Chandrasekaran, J., Feng, H., Lei, Y., Kuhn, D. R., & Kacker, R. (2017, March). Applying combinatorial testing to data mining algorithms. In 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 253-261). IEEE.

[7] "Chauffeur-model:" https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/chauffeur, Accessed: 2020-09-12

[8] Chen, T. Y., Cheung, S. C., & Yiu, S. M. (2020). Metamorphic testing: a new approach for generating next test cases. arXiv preprint arXiv:2002.12543.

[9] Chen, Y., Wang, Z., Wang, D., Fang, C., & Chen, Z. (2019, April). Variable strength combinatorial testing for deep neural networks. In 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 281-284). IEEE.

[10] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 23(7):437–444, July 1997

[11] Dreossi, T., Ghosh, S., Sangiovanni-Vincentelli, A., & Seshia, S. A. (2017). Systematic testing of convolutional neural networks for autonomous driving. arXiv preprint arXiv:1708.03309.

[12] ETAS GmbH. SCODE-ANALYZER Software for describing and visualizing complex closed-loop control systems, 2019. https://www.etas.com/scode.

[13] Gambi, A., Mueller, M., & Fraser, G. (2019, July). Automatically testing self-driving cars with search-based procedural content generation. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 318-328).

[14] Gladisch, C., Heinzemann, C., Herrmann, M., & Woehrle, M. (2020). Leveraging combinatorial testing for safety-critical computer vision datasets. In Proceedings of the

IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (pp. 324-325).

[15] Haq, F. U., Shin, D., Nejati, S., & Briand, L. C. (2020, October). Comparing Offline and Online Testing of Deep Neural Networks: An Autonomous Car Case Study. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST) (pp. 85-95). IEEE.

[16] Herbold, S., & Haar, T. (2020). Smoke Testing for Machine Learning: Simple Tests to Discover Severe Defects. arXiv preprint arXiv:2009.01521.

[17] Jacek Czerwonka. Pairwise testing in real world. In 24th Pacific Northwest Software Quality Conf., volume 200, 2006.

[18] Keras-team/Keras: Deep Learning for humans: https://github.com/keras-team/keras, Accessed: 2020-10-16

[19] Kim, J., Feldt, R., & Yoo, S. (2019, May). Guiding deep learning system testing using surprise adequacy. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (pp. 1039-1049). IEEE.

[20] Komanda model weight file: https://s3.amazonaws.com/udacity-sdc/steering-models/komanda/udacity-challenge2-model/FINE_TUNE_2-checkpoint-sdc-ch2-epoch5, Accessed: 2020-10-19

[21] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010). Practical combinatorial testing. NIST special Publication, 800(142), 142.

[22] Li, Y., Tao, J., & Wotawa, F. (2020). Ontology-based test generation for automated and autonomous driving functions. Information and software technology, 117, 106200.

[23] Ma, L., Juefei-Xu, F., Xue, M., Li, B., Li, L., Liu, Y., & Zhao, J. (2019, February). Deepct: Tomographic combinatorial testing for deep learning systems. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 614-618). IEEE.

[24] Ma, L., Zhang, F., Sun, J., Xue, M., Li, B., Juefei-Xu, F., ... & Wang, Y. (2018, October). Deepmutation: Mutation testing of deep learning systems. In 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE) (pp. 100-111). IEEE.

[25] Majumdar, R., Mathur, A., Pirron, M., Stegner, L., & Zufferey, D. (2019). Paracosm: A language and tool for testing autonomous driving systems. arXiv preprint arXiv:1902.01084.

[26] Odena, A., Olsson, C., Andersen, D., & Goodfellow, I. (2019, May). Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In International Conference on Machine Learning (pp. 4901-4911).

[27] Pei, K., Cao, Y., Yang, J., & Jana, S. (2017, October). Deepxplore: Automated whitebox testing of deep learning systems. In proceedings of the 26th Symposium on Operating Systems Principles (pp. 1-18).

[28] R. Bryce, C. J. Colbourn, M.B. Cohen, "A framework of greedy methods for constructing interaction tests," Proceedngs of the 27th International Conference on Software Engineering (ICSE), pp. 146- 155, 2005

[29] "Rambo-model:" https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/rambo, Accessed: 2020-09-12

[30] Rwightman-model script file: https://github.com/udacity/self-driving-car/blob/master/steering-models/evaluation/rwightman.py, Accessed: 2021-01-20.

[31] Self-driving-car-Results-Dropbox, https://tinyurl.com/y2s6qxeo, Accessed: 2021-01-20.

[32] Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., & Kroening, D. (2018, September). Concolic testing for deep neural networks. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering(pp. 109-119).

[33] Team, T. T. D., Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., ... & Belopolsky, A. (2016). Theano: A Python framework for fast computation of mathematical expressions. arXiv preprint arXiv:1605.02688.

[34] Tesla driver dies in first fatal crash while using autopilot mode: https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk, Accessed: 2020-10-19.

[35] Testing-AI-Systems, https://github.com/cjaganmohan/Testing-AI-Systems, Accessed: 2021-01-20.

[36] Tian, Y., Pei, K., Jana, S., & Ray, B. (2018, May). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In Proceedings of the 40th international conference on software engineering (pp. 303-314).

[37] Tuncali, C. E., Fainekos, G., Ito, H., & Kapinski, J. (2018, June). Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In 2018 IEEE Intelligent Vehicles Symposium (IV) (pp. 1555-1562). IEEE.

[38] Uber's self-driving car didn't know pedestrians could jaywalk: https://www.wired.com/story/ubers-self-driving-car-didnt-know-pedestrians-could-jaywalk/, Accessed: 2020-09-27

[39] "Udacity Challenge 2 – Test Dataset:" https://github.com/udacity/self-driving-car/blob/master/challenges/challenge-2/CH2_final_evaluation.csv, Accessed: 2020-09-12

[40] "Udacity self-driving challenge 2," https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models, Accessed: 2020-09-12

[41] Wicker, M., Huang, X., & Kwiatkowska, M. (2018, April). Feature-guided black-box safety testing of deep neural networks. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 408-426). Springer, Cham.

[42] Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., ... & See, S. (2019, July). Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 146-157).

[43] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. Software Testing, Verification and Reliability, 18(3):125–148, 2008.

[44] Yan, M., Wang, L., & Fei, A. (2019). ARTDL: Adaptive Random Testing for Deep Learning Systems. IEEE Access, 8, 3055-3064.

[45] Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2020). Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering.

[46] Zhang, M., Zhang, Y., Zhang, L., Liu, C., & Khurshid, S. (2018, September). DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 132-142). IEEE.

[47] Zhou, H., Li, W., Zhu, Y., Zhang, Y., Yu, B., Zhang, L., & Liu, C. (2018). Deepbillboard: Systematic physical-world testing of autonomous driving systems. arXiv preprint arXiv:1812.10812.

[48] Zhou, Z. Q., & Sun, L. (2019). Metamorphic testing of driverless cars. Communications of the ACM, 62(3), 61-67.

[49] 2015. Open Source Computer Vision Library. https://github.com/itseez/opencv (2015).

Chapter 5. Evaluation of T-Way Testing of DNNs in Autonomous Driving Systems

The chapter contains a paper accepted at the IEEE 3rd International Conference on Artificial Intelligence Testing (AITest) in 2021.

Evaluation of T-Way Testing of DNNs in Autonomous Driving Systems[*]

Jaganmohan Chandrasekaran[1], Ankita Ramjibhai Patel[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2]

[1]Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, USA

Abstract— A Deep Neural Network (DNN) model is used to perform intelligent, safety-critical tasks in Autonomous Driving Systems (ADS). In our prior work, we proposed a combinatorial testing approach to test DNN models used to predict a car's steering angle. We generate test images by applying a set of combinations of basic image transformations. In this paper, we report a preliminary study that compares the performance of synthetic images generated using a combinatorial approach to DeepTest, a state-of-the-art tool that aims at generating test inputs that maximize neuron coverage. We present an experimental evaluation by measuring and comparing the neuron coverage achieved using the two approaches. Two pre-trained DNN models from the Udacity driving challenge are used as the subject DNNs. The results suggest that the combinatorial approach performs better than the DeepTest approach in generating valid synthetic images and covering an additional number of neurons.

Keywords— Combinatorial Testing, DeepTest, Neuron Coverage.

5.1. INTRODUCTION

Deep Neural Network (DNN) models are used in autonomous driving systems to perform tasks such as pedestrian detection, steering control, object detection. Despite its promising potential, when applied in real-world conditions, the DNN models exhibit erroneous behavior resulting in life-threatening consequences [2]. It is vital to rigorously test these models before their deployment in the real world.

Our earlier work presented a combinatorial approach to generate synthetic images to test the pre-trained DNN models used in self-driving cars [1]. This paper reports two significant extensions of our earlier work. First, in addition to the neuron coverage results reported in [1], we report the neuron coverage for the Chauffeur model. Second, we present a comparative evaluation where we compare the neuron coverage results achieved by our approach to those achieved by DeepTest, a test generation approach that aims at generating test inputs that maximize the neuron coverage [4]. Neuron coverage is a measure of the proportion of neurons activated in a DNN model. Experimental results suggest that in most cases, t-way synthetic images cover an additional number of neurons compared to the DeepTest approach. The remainder of the paper is organized as follows. Section 5-2 presents a brief introduction to the t-way testing of DNNs. Section 5-3 presents the experimental design, results, and discussion. In Section 5-4, we present the concluding remarks and directions for future work.

5.2. T-WAY TESTING OF DNNS

We presented a combinatorial approach to generate t-way synthetic images to test DNN models [1]. In this approach, First, we identify a set of valid image transformations applicable to the seed image. Next, we design an input parameter model (IPM) based on the valid transformations; each valid transformation is mapped as a parameter in the IPM. Then, based on

the IPM, we generate an abstract t-way (t=2) test set. Each t-way test represents a combination of image transformations. Finally, using an image processing library, we generate synthetic images by applying the t-way image transformations to the seed image. The t-way synthetic images are used to test the DNN models.

5.3. EXPERIMENTS

5.3.1. EXPERIMENTAL DESIGN

Tian et al. evaluated the impact of synthetic images generated by combining different image transformations on the neuron coverage using three open-source DNN models, namely Rambo, Chauffeur, and Epoch [4]. In the case of Epoch, a pre-trained model is not publicly available for download. Therefore, we used the remaining two models, namely Rambo and Chauffeur, in this comparison study.

In their evaluation, they generated synthetic images using two approaches, namely *Cumulative transformations* and *Guided transformation*. Similar to our earlier work [1], the guided transformation approach generates synthetic images by combining a set of image transformations. However, this approach aims to generate tests that maximize the neuron coverage and does not guarantee to generate valid synthetic images. That is, while the synthetic images generated using the guided transformation approach can cover an additional set of neurons, they may not be used to determine the correctness of a DNN model because invalid images may never exist in reality.

Therefore, we compare the cumulative neuron coverage achieved by t-way synthetic images to those synthetic images generated using the cumulative transformation approach. We will refer to the cumulative transformation approach as the *DeepTest* approach unless otherwise specified.

To generate synthetic images using the DeepTest approach, we apply a set of valid image transformations identified for the respective seed image. We observed that in most cases, the number of synthetic images generated using a t-way test set is substantially higher compared to that of the DeepTest approach. Therefore, to facilitate a fair comparison, for each group, using a random sampling approach, we select a subset from the t-way test set (synthetic images) such that the number of the t-way tests in the subset is equal to the total number of synthetic images generated using the DeepTest approach.

Then, we execute the DNN model with the seed image (baseline), followed by t-way synthetic images from the subset, and measure the cumulative neuron coverage. We compare the cumulative neuron coverage achieved by the t-way subset with the synthetic images generated using the DeepTest approach. To reduce variations in random sampling, we generated five samples for each group by using different seeds (selected at random).

We refer the reader to our earlier work [1] for additional information about the measurement of cumulative neuron coverage, the number of seed images, the number of valid transformations, and the number of t-way test cases generated for each seed image.

5.3.2. Results and Discussion

First, we present the cumulative neuron coverage achieved by t-way tests for Chauffeur. The Chauffeur model consists of 1 CNN sub-model with 1427 neurons and 1 LSTM sub-model with 513 neurons. Tian et al. did not include the LSTM sub-model in their evaluation. Hence, for Chauffeur, we limit our comparison to the CNN sub-model.

For the Chauffeur model, 14 out of 19 seed images cover less than 15% of the total neurons (1427 neurons); Among the seed images, Group 7 covers the least, covering 6% of total neurons

(90 neurons), while the seed image from Group 16 covers the most with 22% of total neurons (318 neurons).

Figure 5-1 presents the cumulative neuron coverage achieved by t-way tests for Chauffeur. The x-axis represents the group number. The y-axis represents the percentage of additional neurons covered by the t-way tests compared to their respective baseline. Our results suggest that t-way tests result in a significant increase in neuron coverage. Out of nineteen groups, t-way tests generated for sixteen groups achieve more than one hundred percent increase in cumulative neuron coverage.

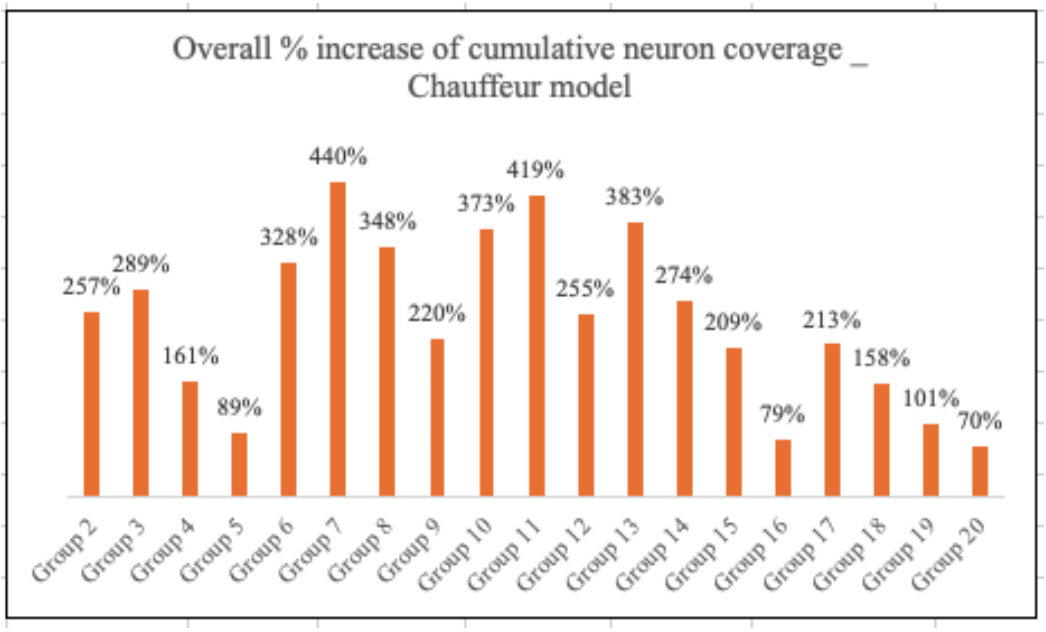


FIGURE 5-1 CUMULATIVE COVERAGE – CHAUFFEUR

Next, we present the comparison results. For Rambo, the coverage results obtained from our earlier work are re-used in our comparison experiments. Figure 5-2 and Figure 5-3 present the comparison results for Rambo and Chauffeur, respectively. The x-axis represents the group number. The y-axis represents the number of neurons. Due to space limitations, we present the average cumulative neuron coverage achieved by the five t-way subsets for each group. A

horizontal blue bar in the bar chart indicates the cumulative neuron coverage achieved using the DeepTest approach. Our results indicate that for Rambo, in most cases (18 out of 19 groups), subsets of the t-way test set achieve a higher cumulative coverage compared to the DeepTest approach. For five groups (Group 2, 7, 8, 9, 13), the subset (of the t-way test) covers a significant number of additional neurons compared to the DeepTest approach.

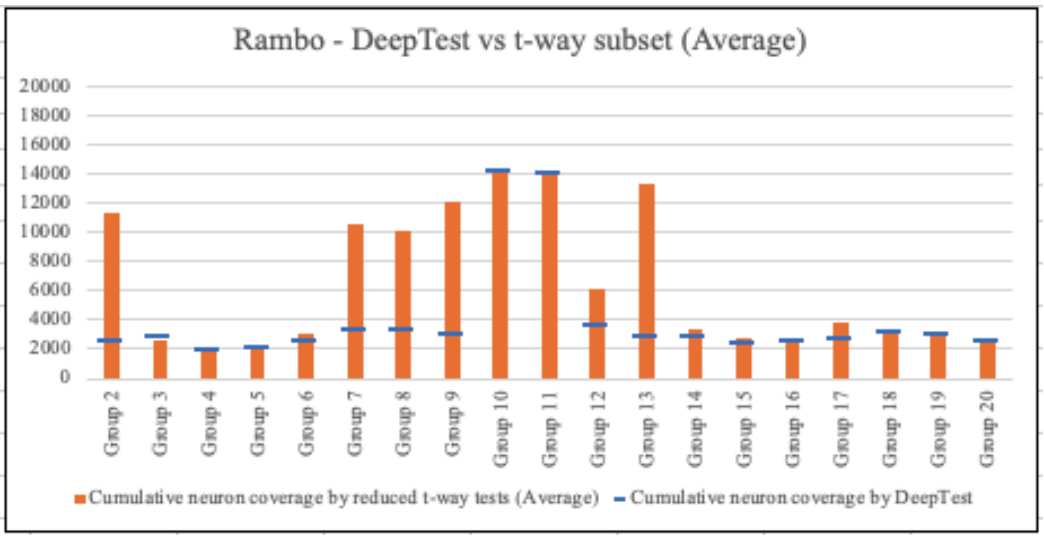


FIGURE 5-2 COMPARISON – DEEPTEST VS REDUCED T-WAY (RAMBO)

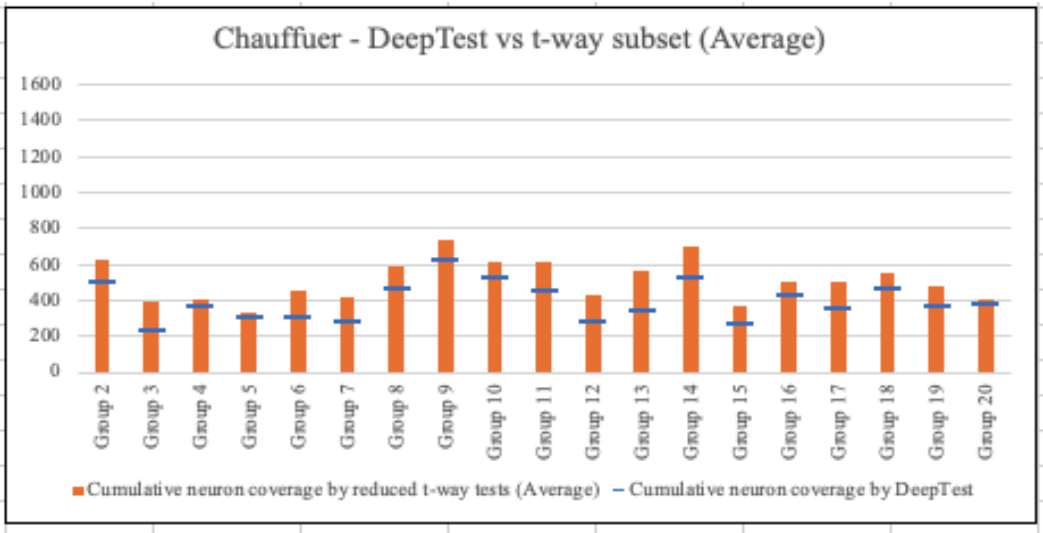


FIGURE 5- 3 COMPARISON – DEEPTEST VS REDUCED T-WAY (CHAUFFEUR)

In the case of Chauffeur, for 16 groups, all five samples of the t-way subset cover a significant additional number of neurons compared to the DeepTest approach. For the remaining three groups (Group 4, 5, 20), the t-way subset covers a marginally higher number of neurons than the DeepTest approach.

Overall, the results from this initial study indicate that synthetic images generated using the combinatorial approach can achieve higher neuron coverage than the DeepTest approach. The source code, results, data and/or artifacts have been made available at [3].

5.4. Conclusion and Future work

In this paper, we present the cumulative neuron coverage for the Chauffeur model and an initial study that compares the synthetic images generated using a combinatorial approach to that of DeepTest in terms of cumulative neuron coverage. In most cases, the results suggest that the synthetic images generated using the combinatorial approach cover an additional number of neurons compared to the DeepTest approach.

As part of future work, we plan to conduct a comprehensive empirical study that compares the effectiveness of combinatorial testing to that of random testing in testing pre-trained DNN models.

5.5. Acknowledgment

This work is supported by research grant (70NANB18H207) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

5.6. References

[1] Chandrasekaran, J., Lei, Y., Kacker, R., & Kuhn, D. R. (2021, April). A Combinatorial Approach to Testing Deep Neural Network-based Autonomous Driving Systems. In *2021*

IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 57-66). IEEE.

[2] Tesla driver dies in first fatal crash while using autopilot mode: https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk, Accessed: 2021-07-10.

[3] Testing-AI-Systems, https://tinyurl.com/k4aswyef

[4] Tian, Y., Pei, K., Jana, S., & Ray, B. (2018, May). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering* (pp. 303-314).

Chapter 6. A Combinatorial Approach to Explaining Image Classifiers

The chapter contains a paper published in IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2021

A Combinatorial Approach to Explaining Image Classifiers*

Jaganmohan Chandrasekaran[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2]

[1]Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, USA

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, USA

***Abstract*—**Machine Learning (ML) models, a core component to artificial intelligence systems, often come as a black box to the user, leading to the problem of interpretability. Explainable Artificial Intelligence (XAI) is key to providing confidence and trustworthiness for machine learning-based software systems. We observe a fundamental connection between XAI and software fault localization. In this paper, we present an approach that uses BEN, a combinatorial testing-based software fault localization approach, to produce explanations for decisions made by ML models.

Keywords— explainability, deep learning, software testing, debugging DNN models, explainable AI, combinatorial testing, Image classifiers, model-agnostic, counterfactual explanation, instance-level explanations

6.1. INTRODUCTION

Artificial Intelligence (AI) based software systems are increasingly adopted in safety-critical domains, e.g. medical imaging and autonomous driving. At the core of AI-based software systems is a machine learning (ML) model that is used to perform tasks such as classification and

prediction. The ML models used in such tasks are black box in nature, i.e., the reasoning behind their decision is typically not known to the user. Using a black box model in AI software systems could compromise trustworthiness and create problems such as racial and gender bias. There is an urgent need to provide explanations for the decisions made by AI-based software systems.

Explainable Artificial Intelligence (XAI) focuses on creating approaches and tools that can automatically provide explanations for the decisions made by ML models [3]. In particular, XAI tries to answer the following two questions: Why does the model make a particular decision? What are the major factors that contribute to the decision? XAI has attracted a lot of interest from both academia and industry in the past few years. Providing explanations allows a model to be interpreted, which is key to acceptance of AI technologies. Furthermore, information gathered from an interpretable model can help engineers determine the cause of incorrect decisions.

There are two types of explanations for AI decisions. *Local explanations* are created to explain a specific decision, whereas *global explanations* are created to explain an entire model. In this paper we present an approach that creates local explanations using the counterfactual approach, which human factor studies have shown to be highly effective for explanation [11]. A counterfactual approach tries to identify a minimum set of features that, if removed, would cause a different decision to be made [9].

The key insight is that from an abstract perspective, producing a counterfactual explanation for a local decision made by an ML model is similar to the fault localization problem [15][16]. In fault localization, given a failing scenario, a software developer identifies which part of the input that causes the failure. Similarly, in XAI, given a decision made by an ML model, we identify features that causes the decision, in the sense that if these features are removed, then the decision would be different.

Specifically, we explore the use of a combinatorial testing-based fault localization approach called BEN to produce counterfactual explanations for image classifiers. Given a t-way test set, BEN identifies a failure-inducing (or inducing) combination that causes every test (for a deterministic system) containing the combination to fail and that is as small as possible [8]. We apply BEN to quickly identify a minimal subset of features in an image that, if removed, would result in a different classification.

Assume that a model M produces a classification X for an input image I. To produce a counterfactual explanation for this classification result, we first perform segmentation on image I. In image segmentation, various algorithms are used to assign a class to each pixel of an image. For example, in a street scene, boundary detection and other algorithms may identify classes “sign”, “human”, “car”, etc., and each pixel of the image is associated with one of the classes. The segmentation process may be applied at a more granular level to identify parts of objects. Each segment is modeled as a Boolean parameter. We build a 2-way test set for these parameters. Each test can be used to derive a test image from the original image, i.e. image I. A segment is masked in the test image if the corresponding parameter is true in the test; otherwise, a segment is retained without modification.

The notion of test execution is mapped to image classification in the following sense. If a test image is classified by model M differently than the original image, the corresponding test execution is considered to be failing. Otherwise, the corresponding test execution is considered to be passing. The 2-way test set with execution statuses is then fed to BEN to identify inducing combinations. In the identification process, BEN could generate additional tests, which can be executed in the same manner. That is, for each additional test, a test image is first derived and then classified using model M to determine its execution status.

Finally, each inducing combination identified by BEN is used to derive an image that produces a different classification. This image serves as a counterfactual explanation for the original classification X.

We report an experimental evaluation of our approach. We use the *VGG16* model [25], a popular image classifier as our subject model and fifty randomly selected seed images from the ImageNet test dataset [23]. Our results suggest for 44 (out of 50) images, our approach can generate counterfactual explanations. Furthermore, in most cases, our approach can generate a counterfactual explanation by removing no more than two segments from the input image.

The remainder this paper is organized as follows. Section 6-2 provides an introduction to Deep Neural Network-based image classifiers, counterfactual explanations, and BEN. In Section 6-3, we present our approach and give an example to illustrate the approach. Section 6-4 reports the experimental evaluation of our approach, where we present our experimental design, results and discussion. Section 6-5 discusses the existing work on XAI. Section 6-6 provides concluding remarks and directions for our future work.

6.2. BACKGROUND

6.2.1. DEEP NEURAL NETWORKS

Deep learning is used across domains such as autonomous driving, speech recognition, speech translation, and medical imaging. At the core of deep learning is a Deep Neural Network (DNN) that is used to perform tasks such as image classification, object detection, and others. A DNN follows a neural network architecture and consists of an input layer, several hidden layers and an output layer. A trained DNN model takes an input (e.g., an image) and produces a prediction as output.

Compared to traditional software development, where the programming logic is implemented based on rules derived from the requirements, DNN based applications derive their decision logic (learning) from a training dataset. The decision logic is referred to as the trained DNN model.

In recent years, deep learning-based image recognition software systems have improved significantly and could be more efficient than humans in some domains. A practitioner can build a DNN model using different types of neural network architecture. One of the popular neural network architectures used for image recognition tasks is convolutional neural networks (CNN). Given an input, CNN architecture is known for its ability to detect important features without any human supervision. The subject models used in our experiments use a CNN based architecture and perform image classification.

6.2.2. AI EXPLANATIONS

The explanations generated by XAI tools can be categorized into two types, feature-importance based explanations and counterfactual explanations. Assume a model M that produces a classification X for an input image I. A feature-importance based explanation identifies a set of important features of I that contribute to decision X. In addition, it assigns weights to the features that quantify their contribution. In contrast, a counterfactual explanation identifies a minimum set of features of I that if removed, shall change the prediction. In other words, counterfactual explanations are contrastive in nature.

6.2.3. BEN

Ghandehari et al. developed a combinatorial testing-based approach called BEN to software fault localization [5, 6]. Localizing a fault using BEN consists of two major phases:

inducing combination identification (Phase I) and faulty statement localization (Phase II). BEN assumes that a combinatorial t-way test set is available and has been executed on the system under test (SUT). In the first phase, BEN takes the t-way test set and its results as input and tries to identify one or more inducing combinations in an iterative manner. BEN analyses the test file and identifies a set of t-way suspicious combinations. Based on the t-way suspicious combination(s), BEN generates a new t-way test set. For the new t-way test set, the user generates concrete tests, executes the tests, and records their execution status (either pass or fail). Then, the user provides the execution status back to BEN. This process is repeated until BEN identifies an inducing combination. Note that BEN expects the initial test set to contain at least one passing and one failing test. If there is no passing test in the initial t-way test set, BEN identifies an inducing combination based on the initial t-way test set. In our approach, the inducing combination identified by BEN is used to generate counterfactual explanations. Phase II of BEN is not utilized in our approach.

6.3. APPROACH

This section presents a combinatorial approach to generate counterfactual explanations for machine learning models that take an image as input and output a prediction. Our approach consists of four phases: Image segmentation, t-way testing, identifying inducing segments, and constructing explanations.

Image Segmentation: Image segmentation is a widely used image processing technique that partitions a digital image into different segments based on the characteristics of the image pixels. In our approach we first perform image segmentation on the input image. As discussed later, each segment is modeled as a parameter during combinatorial testing. Working with

segments instead of pixels allows us to reduce the number of parameters in our input parameter model (IPM).

We point out that the number of segments could potentially affect the quality of the counterfactual explanation. The more segments, the finer grained the resulting explanation could be. However, the more segments, the more parameters, the more expensive to produce the explanation. Many segmentation algorithms allow the user to define a maximum number of segments. The exact number of segments produced by the segmentation process is typically close to the maximum number. A trade-off decision often needs to be made when choosing the maximum number of segments.

Recall that BEN assumes that there exists an input parameter model (IPM) of the SUT, a test oracle to determine the status of the test execution, and a t-way combinatorial test set with execution results. In the following we discuss how to provide these components in the context of XAI.

T-Way Testing: We begin this phase by deriving an input parameter model for the SUT (input image). For an input image, every segment is considered as a parameter.

Our approach aims to identify a minimum number of segments that, if removed, would change the prediction. To remove a segment, we perform a masking operation on the particular segment. In our approach, a segment can either be masked or not masked. Therefore, in the IPM, for each parameter, we identify the following two values – true (masked) and false (not masked).

Then, we generate an abstract t-way test set using ACTS, a combinatorial test generation tool [2]. We derive the concrete tests by applying masking to specific image segments (as per the test case) using image-processing python libraries [1, 18, 24, 30]. We execute the concrete tests (images) and determine their execution statuses.

Given an image, the DNN model produces a class label (prediction) as output. To determine the execution status of a test, we define the test oracle as follows: On executing the model with a test image, if the output (class label) matches that of the original image, we consider it to be a passing test. If the output does not match the output of the original image, we consider it to be a failing test.

Identifying Inducing Combinations: We begin this phase by providing an initial test file (as input) to BEN. The initial test file includes parameters and values, the test strength, the initial t-way test set, and the execution status of each test. In each iteration, analyzing the test file, BEN either generates an additional set of tests or terminates by identifying inducing combination(s). For additional tests generated by BEN, we derive concrete tests (t-way images), execute the model with the test images and update their execution statuses. Then, we provide the updated test results to BEN.

This process continues until one of the stopping conditions is satisfied: (1) an inducing combination is identified by BEN, or (2) the user decides to stop the process. In the latter case, the top-ranked suspicious combination is considered to be the inducing combination, and we proceed to the next phase.

Constructing Explanations: In this phase, we derive explanations based on the inducing combinations in an iterative manner.

Given the nature of the XAI problem, an inducing combination identified by BEN may not be directly used to produce a counterfactual explanation. Consider a scenario where an input image has 20 segments (i.e., 20 parameters, and each parameter has two values - TRUE, FALSE). BEN identifies the following two inducing combinations: (segment_1 = FALSE, segment_4 = FALSE), (segment_2 = TRUE, segment_4 = FALSE).

The first inducing combination suggests a test retaining segment_1 and segment_4 shall fail (change the prediction). Even though all the test images that contain these two segments have a different classification, this inducing combination cannot be used to produce a counterfactual explanation, since it does not suggest any segments to be removed.

The second inducing combination suggests to remove segment_2 (masked) while retaining segment_4 in order to produce a different classification. This combination can be used to produce a counterfactual explanation as discussed next.

In general, an inducing combination that suggests the removal of one or more segments can be used to produce counterfactual explanations.

We begin to construct a counterfactual explanation by selecting the top-ranked inducing combination, generating an image based on the inducing combination (modified image), executing the model with the image, and recording its execution status. Suppose the prediction of the modified image differs from the prediction of the original image (fail). In that case, the approach stops, and the modified image is shown as an explanation to the user.

Otherwise, if the prediction of the modified image is the same as the prediction of the original image state (pass), we select the next ranked inducing combination and repeat the process, i.e., generate an image based on the inducing combination, and execute and compare its prediction with the original prediction.

This process is continued until either of the two conditions is satisfied: (1) the prediction of a modified image generated based on inducing combination(s) differs from the prediction of the original image; or (2) all the modified images generated based on the inducing combination(s) match the original prediction. In the first case, the modified image is shown as an explanation to the user. In the second case, we derive an explanation as follows.

First, we analyze the test suite and identify a test that (1) contains the inducing combination, and (2) the prediction differs from the original prediction (i.e., a failing test). If there is more than one test that satisfies the two criteria, we select a test with the least number of masked segments. Recall that our objective is to identify a minimal number of segments that, if removed, shall change the prediction.

Next, in addition to the inducing combination, we mask the additional segments whose values are true in the test in an incremental manner (one segment at a time), starting with the segments closer to the segments in the inducing combination. This process is repeated until the prediction of the modified image differs from the original prediction. The modified image is shown as an explanation to the user. Note that masking additional segments from a failing test is likely to produce a counterfactual explanation, since its prediction differs from the original prediction.

Example: We illustrate our approach using an example. Consider the image in Figure 6-1. It is assumed that the DNN model is executed with the image and the prediction result (P) is available.

To derive a counterfactual explanation, we begin with image segmentation, which identified the possible number of segments for the subject image as 20 (Figure 6-2).

Next, we build an IPM with 20 parameters; each parameter has two values: {TRUE and FALSE}. Then, we generate a 2-way test set (12 tests) using ACTS [2]. We derive the concrete tests (test images), execute the model with concrete tests, record and compare their execution statuses (P`) with the original prediction (P). Based on the execution statuses, we have four passing tests (P = P`) and eight failing tests (P != P`). A test file is generated, and it contains the IPM, the strength of the t-way test set, the t-way test set, and its execution status.

Next we provide the test file as input to BEN. After a couple of iterations, BEN identifies an inducing combination - *segment_10=TRUE,segment_12=TRUE,segment_17=false*. Note that at each iteration, we repeat the process of deriving, executing, and updating the status of the additional tests.

To derive a counterfactual explanation, we generate a modified image based on the inducing combination - *segment_10=TRUE,segment_12=TRUE,segment_17=FALSE.*

Although the inducing combination consists of three segments, the modified image will have two (out of three) segments, namely *S_10, S_12* masked, while no changes being made to *S_17*, as its value is *FALSE, i.e., not to mask the segment.* Then, we execute the model with the modified image, and its output (prediction) is compared to the output of the original image. The prediction of the modified image differs from the original prediction.

At this point, the approach terminates, and the modified image (Figure 6-3) is shown as a counterfactual explanation to the user.

Original image

Prediction: **pomegranate**

FIGURE 6-1 ORIGINAL IMAGE

Segmentation

Number of segments = **20**

FIGURE 6-2 SEGMENTATION OF INPUT IMAGE

Counterfactual explanation

Masked segments = 10, 12

Prediction: **hip**

FIGURE 6-3 COUNTERFACTUAL EXPLANATION

6.4. EXPERIMENTS

In this section, first, we present the design of our experiments including the research question, the subject model and selection of seed images, segmentation and masking techniques, and the metrics used to measure the effectiveness of our approach. Second, we present and discuss our results. Third, we compare the results of our approach with SHAP, a popular state-of-the art XAI tool. Finally, the threats to validity are discussed. The source code, data and/or artifacts have been made available at [28, 29]

6.4.1. RESEARCH QUESTIONS

The major research question of our evaluation is the following:

- How effective is BEN in generating counterfactual explanations for DNN-based image classifiers?

6.4.2. MODEL

We evaluate our approach using an open-source, pre-trained model – VGG16 [25]. The model uses a convolutional neural network architecture consisting of 13 convolution layers and three dense layers. VGG16 is used in evaluating similar explainable AI tools [26].

6.4.3. SEED IMAGES

The ImageNet dataset is an extensive collection of visual images. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition for evaluating algorithms for object detection and image classification [23]. The VGG16 model, a runner-up at the ILSVRC 2014 challenge, is trained using the ImageNet dataset with over 14 million images with 1000 classes.

In our experiments, we use the ILSVRC2017 test dataset, the latest test dataset from ImageNet (ILSVRC2017). The test dataset consists of 5500 images [13, 23]. We randomly selected fifty seed images.

6.4.4. SEGMENTATION

The Simple Linear Iterative Clustering (SLIC) algorithm is used to perform image segmentation [1]. Based on the maximum number of desired segments provided by the user, the SLIC algorithm clusters pixels based on their color similarity and proximity in the image plane and create segments. In our experiments, we set the maximum number of segments to 25. However, the exact number of possible segments varies for each seed image. This is because the SLIC algorithm generates segments based on certain properties of an image.

6.4.5. MASKING OF SEGMENTS

An image consists of an array of dots referred to as pixels. Pixels of a color image can have a value in the range of 0 to 255. The value of 0 represents a black pixel, and the value of 255 denotes a white pixel. In our experiments, we mask a segment by setting all its pixels to the value of 0.

6.4.6. METRICS

The effectiveness of our approach is measured in terms of the quality of the counterfactual explanations it produces. The quality of a counterfactual explanation could be measured in different ways [3][19]. Ultimately, a counterfactual explanation should make sense to a human subject. This is however subjective.

In our experiments, the quality of a counterfactual explanation is measured in the following two aspects: (1) the number of segments that need to be removed from the original image to

produce the explanation. The fewer segments to be removed, the easier to be understood, the higher quality. (2) the explanation must produce a different prediction than the original prediction.

6.4.7. Results and Discussion

Our approach effectively derived counterfactual explanations for 44 (out of 50) seed images. In the following, we present the details of our results. Due to space limitations, we only show some example results in this section. The complete results are available at [28, 29].

6.4.7.1. Counterfactual Explanations

First, we present the results of counterfactual explanations generated from an inducing combination alone i.e., no additional segments need to be removed. For 24 out of 50 respective inducing combination (identified by BEN) effectively change the original prediction.

Our results show that for 6 out of 24 images, our approach removes one segment to produce the counterfactual explanation. For 16 out of 24 images, our approach only removes 2 segments. For the remaining 2 out of 24 images, our approach removes three segments.

Figure 6-4 shows some example results of these images. In each row, the first image is the original seed image; the second image shows the segmentation applied to the seed image. The third image is the counterfactual explanation. For the image in Row 1, removing one segment (segment 4) modifies the prediction from white_stork to black_stork. For the images in Row 2 (original prediction: dragonfly) and Row 3 (original prediction: stage), removing two segments changes the prediction to lycaenid and feather_boa, respectively. For the image in Row 4 removing 3 segments changes the prediction from sea lion to promontory.

Original Image

Prediction: *white_stork*

Segmentation

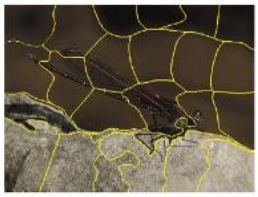
Number of segments = **20**

Counterfactual Explanation

Prediction: *black_stork*

Original Image

Prediction: *dragonfly*



Segmentation

Number of segments = **23**

Counterfactual Explanation

Prediction: *lycaenid*

Original Image

Prediction: *stage*

Segmentation

Number of segments = **17**

Counterfactual Explanation

Prediction: *feather_boa*

Original Image

Prediction: *sea lion*

Segmentation

Number of segments = **20**

Counterfactual Explanation

Prediction: *promontory*

FIGURE 6-4 COUNTERFACTUAL EXPLANATIONS DERIVED FROM INDUCING COMBINATIONS

Next, we discuss the counterfactual explanations that cannot be derived from the inducing combination alone. Instead, some additional segments need to be removed to produce a counterfactual explanation. For 20 images in our experiments, the modified images generated from their respective inducing combinations alone do not change the predicted class labels. Therefore, additional segments must be removed for these images in order to produce a counterfactual explanation.

Our results indicate that for 8 out of 20 images, masking one additional segment along with the inducing combination was sufficient to change the classification. For 7 out of the remaining 15 images, two additional segments needed to be masked. For the remaining 5 images, in addition to the inducing combinations, we masked three to five additional segments to generate a counterfactual explanation.

Figure 6-5 presents some of the counterfactual explanations generated from the inducting combination and one or more additional segments. In each row, the first image is the original seed image, followed by the segmentation applied to the seed image and the modified image produced based on their respective inducing combination. The fourth image is the counterfactual explanation produced from the inducing combination and one or more additional segments.

For the image in Row 1 - image #2737 with an original prediction - mountain_bike, masking one segment (segment_4=true), in addition to the inducing combination (segment_5=true, segment_13=true), changes the original prediction from mountain_bike to moped. Similarly, for image #4148 (Row 2) with an original prediction of Arabian_camel, masking one additional segment changes the original prediction from Arabian_camel to a sarong.

FIGURE 6-5 COUNTERFACTUAL EXPLANATIONS DERIVED FROM INDUCING COMBINATIONS AND ADDITIONAL SEGMENTS

Consider the image in Row 3 (image #3793, original prediction – tiger), in addition to the inducing combination (segment_7=true, segment_19=true), masking four more segments (segment 2, 4, 17, 20) is necessary to change the original prediction from tiger to an Egyptian_cat.

The results suggest that in most cases, our approach can efficiently generate a high-quality counterfactual for image classifiers. In other words, our approach can effectively identify a minimal (2 or 3 segments) yet important set of segments that if removed, would modify the original prediction.

We note that BEN was unable to identify inducing combinations for six seed images. For one of the seed images (image # 4541), BEN terminated with an error message. There is no suspicious combination whose length is 2. For the remaining five seed images, in spite of multiple iterations, BEN failed to identify an inducing combination. We observe that all the additional tests generated by BEN resulted in a passing status for each of these images. Therefore, we suspect BEN is unable to find an inducing combination as it expects at least one failing test to identify an inducing combination. We plan to investigate this as part of future work.

6.4.7.2. COMPARISON WITH SHAP

We compare the counter-factual explanations (derived by our approach) with SHAP, a widely used feature-importance approach tool [17]. Given an input and a pre-trained model, SHAP produces explanations for a model's decision by ranking the input features that contributed to the model's decision (feature-importance-based explanation). This comparison allows us to see the importance of the segments removed by our approach to produce a counterfactual explanation.

Figure 6-6 presents some of the comparison results. The first image in each row presents the counterfactual explanation identified by our approach. The second image represents the output produced by the SHAP tool. SHAP output consists of four images: the original image (provided

as input to the SHAP algorithm), followed by the top three predictions from the model with the features (segments) contributing to that corresponding predictions. Features (segments) that positively contribute to the outcome are highlighted in green, and features (segments) that negatively contribute to the outcome are highlighted in red [14].

Among the five images, the output from SHAP suggests, the set of segments that are removed to generate a counterfactual explanation in our approach positively contributes to the original decision (highlighted in green color). In other words, our approach identifies a minimal yet significant set of segments that if removed, shall modify the prediction. One of the interesting examples is the image from row 3 in Figure 6-6. The image consists of two performers, a microphone and a guitar. The predicted class label for the original seed image is a *stage*. Our approach suggests a part of a performer's body (segment #7) be removed to generate a counterfactual explanation, which is an unexpected segment (intuitively). However, the results from SHAP confirm that segment #7 contributes significantly to the predicted class label.

A counterfactual explanation does not determine the correctness of the original prediction. Consider the image from the image from row 5 in Figure 6-6. The image consists of a person on a motorcycle. The predicted class label for the original seed image (*mountain_bike)* might not match the user's expectation (*motorcycle)*. In such scenarios, i.e., in case of a model's misprediction, deriving a counterfactual explanation can identify a set of features (segments) that if removed would modify the prediction. In other words, a counterfactual explanation could help identify a set of features that contribute to the misprediction. Likewise, SHAP also indicates the set of segments that contribute to the original prediction - mountain_bike. As part of future work, we plan to investigate the possibility of using the feedback from the counterfactual explanations to debug a model's misprediction.

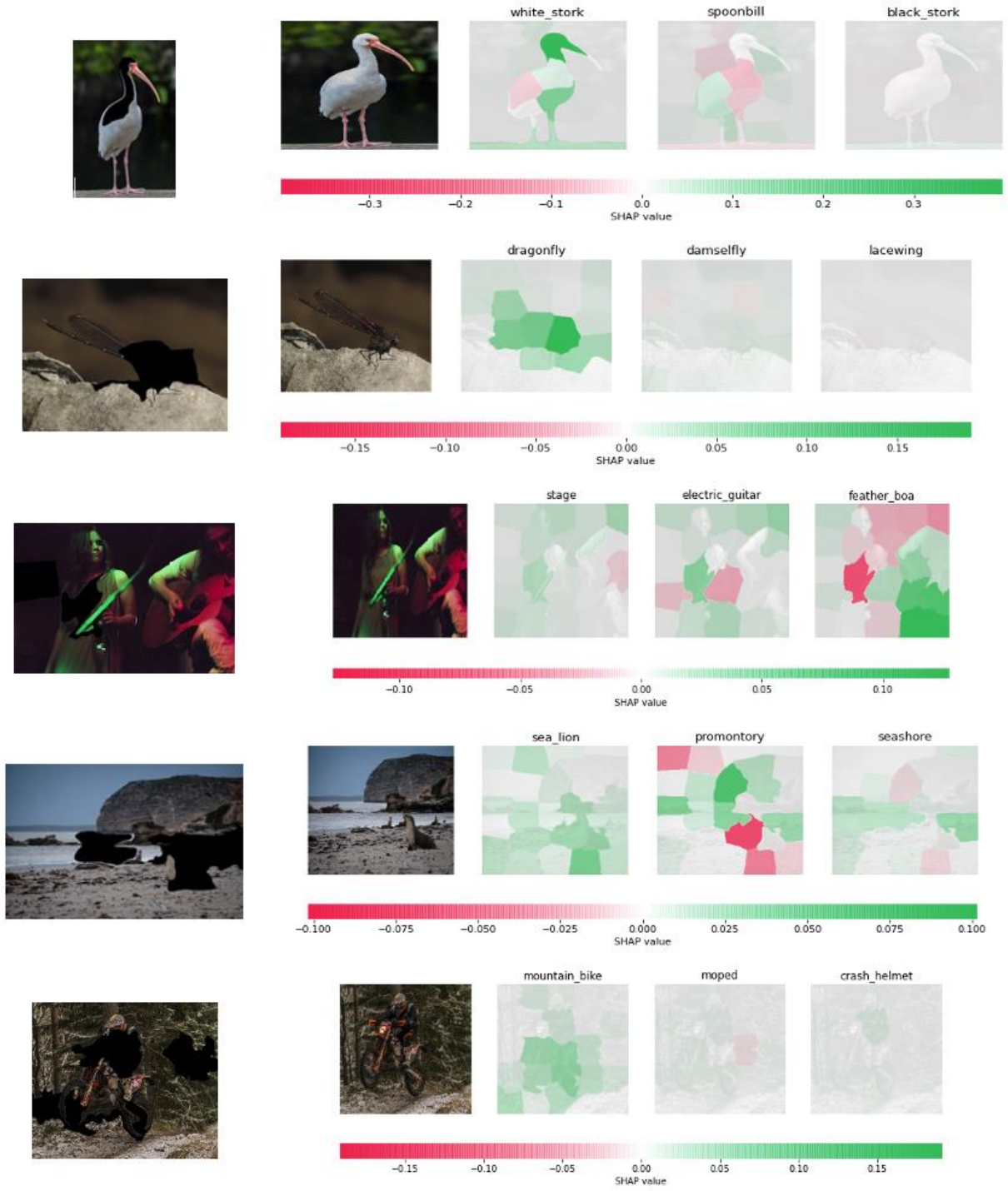


FIGURE 6-6 COMPARISON WITH SHAP

This is an initial comparison study. The overall results indicate that BEN can be effectively adopted to derive a counterfactual explanation that indicates significant segments, i.e., segments that make a significant contribution to a model's decision. SHAP performs 1000 iterations to generate explanations. That is, SHAP executes the VGG16 model 1000 times to identify the important features that contribute to a model's decision. In contrast, our approach derives explanations with an average of 20 - 25 test cases (image perturbations). In other words, we derive a counterfactual explanation by executing the VGG16 model for an average of 25 times. We plan to perform a detailed, comprehensive comparison with other state-of-the-art explainable AI tools as part of future work.

6.4.8. THREATS TO VALIDITY

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. To mitigate the risk of human errors, we tried to automate as many tasks as possible, from generating synthetic images to executing the tests. Also, we have manually checked some of the results whenever any inconsistent or surprising results occur. For example, for image 3456 and 3462 all the initial tests resulted in a failure. In contrast, image 3703 and 3793 had a mix of passing and failing test. In such scenarios, we manually verified test results by inspecting the images and the prediction results.

Threats to external validity occur when the results from our experiments could not be generalized to other subjects. The DNN model architecture used in our study have been used in other studies [26, 27]. We randomly selected fifty seed images from ImageNet, a large, diverse dataset with more than 5000 images. This helps to alleviate the risk of lack of diverse images used in our study.

6.5. Related Work

In this section, we discuss existing work that is closely related to our work. First, we discuss existing work on counterfactual explanations. Dhurandhar et al. proposed a method that produces contrastive explanations. Their method identifies two sets of pixels: (1) A minimal set of features that are sufficient to obtain the current classification (pertinent positive); and (2) A minimal set of features that should be absent to obtain the current classification (pertinent negative) [4]. In contrast, we identify a minimal number of features (segments) that if removed (absent), will change the current classification.

Goyal et al. proposed a technique that generates counterfactual visual explanations [7]. Assume that for an input image I, model M predicts class A. Their approach generates a visual explanation that tries to answer the following question: How should the image I be different for the model to predict Class B instead of Class A? Our work is similar to theirs in terms of altering the input image and showing a modified image as a counterfactual explanation. However, in our approach, the modification is limited to removal of one or more segments from the original image, whereas they generate counterfactual explanation by identifying and replacing regions of the original image with regions from the image belonging to the counterfactual class.

Vermeire et al. proposed a model-agnostic approach to generate counterfactual explanations for image classifiers [27]. Our work is similar to theirs in terms of identifying segments that, if removed, shall change the classification. Our work is different from theirs in the following way: They propose two methods, i.e., Search for Evidence Counterfactuals (SEDC) and Search for Evidence Counterfactuals with Target Counterfactual Class (SEDC-T). SEDC uses a best-first search approach to generate a counterfactual explanation. In SEDC-T, a counterfactual explanation is generated by removing segments (iteratively) to reach a predefined target class. In

contrast, we use a combinatorial testing-based approach to generate counterfactual explanations, and our approach does not generate an explanation for a predefined target class.

Hendricks et al. propose a method that produces a descriptive counterfactual text as an explanation to the end user [10]. Compared to this, our approach displays a modified image (with removed segments) to the end user.

Existing work reported in [8, 20] generates counterfactual explanations for tabular data. In contrast, our work generates counterfactual explanations for an image data.

Riberio et al. proposed LIME that generates local explanations based on input perturbations that probe a ML model and derive explanations [21, 22]. Lundberg et al. proposed SHAP that generates explanations using game theoretic framework [17]. Similar to our work, LIME and SHAP create image perturbations by segmentation to derive an explanation. However, our work focuses on generating a counterfactual explanation, whereas their work focuses on identifying important features that contribute to the original decision.

Sun et al. proposed a statistical fault localization-based approach called DeepCover to generate explanations for image classifiers [26]. In their approach every pixel from the image is assigned a score in terms of their likelihood to contribute to the original decision. An explanation is derived by adding sufficient pixels (a subset of the original pixels) that shall produce the original decision. Our work is similar to their work in terms of using a software fault localization-based approach to derive explanations. However, our work differs in the following two ways: 1) we generate t-way test inputs (image perturbations) whereas their approach randomly selects and masks a set of pixels; and 2) we generate a counterfactual explanation whereas their explanation focuses on the pixels that contribute to the original decision.

Similar to our work, Kuhn et al. adopted a combinatorial fault location process and reported an approach that identifies a unique t-way combination that contributes to a model's decision [15][16]. Their approach is designed for tabular data. In contrast, our approach focuses on image-based classifiers and produces counterfactual explanations.

6.6. Conclusion and Future work

In this paper, we present a combinatorial testing-based approach to explaining image classifiers. Our approach is model-agnostic, as it treats the underlying model as a black box. We evaluated our approach using the VGG16 model [25] and seed images from the ImageNet dataset [23]. Our results suggest that for 44 (out of 50) images, our approach can effectively generate a counterfactual explanation. For 28 images, the counterfactual explanation is generated by removing no more than 2 segments. Overall, the results indicate BEN, a combinatorial testing-based fault localization approach, has the potential to be effectively applied and derive explanations for ML models.

In some cases (6 out 50 images), we are unable to derive counterfactual explanations using our approach. Based on the initial analysis, we suspect that BEN is unable to find an inducing combination as it expects at least one failing test. Therefore, as part of future work, we plan to investigate this by increasing the initial test strength or increasing the segment size or both.

In addition, we plan to continue our work in the following directions. In our current approach, a counterfactual explanation cannot be derived from the inducing combination alone for some images. Also, some inducing combinations suggest no changes to be made, i.e., *not to mask any segment.* First, we plan to investigate how to generate effective inducing combinations. Second, in the case of a model's misclassification, a counterfactual explanation could help identify a set of features that contribute to the misclassification. We plan to investigate how to use the

feedback from the counterfactual explanations for model debugging. Third, we plan to extend this work to generate a counterfactual explanation for ML models trained with tabular data. Finally, we plan to include additional subject models and perform a detailed, comprehensive comparison with similar XAI tools.

6.7. Acknowledgment

This work is supported by research grant (70NANB18H207) from Information Technology Lab of National Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

6.8. References

[1] Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., & Süsstrunk, S. (2012). SLIC superpixels compared to state-of-the-art superpixel methods. *IEEE transactions on pattern analysis and machine intelligence*, *34*(11), 2274-2282.

[2] Advanced Combinatorial Testing System (ACTS), https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software , Accessed: 2021-01-20.

[3] Arrieta, A. B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., ... & Herrera, F. (2020). Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, *58*, 82-115.

[4] Dhurandhar, A., Chen, P. Y., Luss, R., Tu, C. C., Ting, P., Shanmugam, K., & Das, P. (2018). Explanations based on the missing: Towards contrastive explanations with pertinent negatives. In *Advances in neural information processing systems* (pp. 592-603).

[5] Ghandehari, L. S., Chandrasekaran, J., Lei, Y., Kacker, R., & Kuhn, D. R. (2015, April). BEN: A combinatorial testing-based fault localization tool. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 1-4). IEEE.

[6] Ghandehari, L. S., Lei, Y., Kacker, R., Kuhn, D. R. R., Kung, D., & Xie, T. (2018). A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*.

[7] Goyal, Y., Wu, Z., Ernst, J., Batra, D., Parikh, D., & Lee, S. (2019). Counterfactual visual explanations. *arXiv preprint arXiv:1904.07451*.

[8] Guidotti, R., Monreale, A., Ruggieri, S., Pedreschi, D., Turini, F., & Giannotti, F. (2018). Local rule-based explanations of black box decision systems. *arXiv preprint arXiv:1805.10820*.

[9] Guidotti, R., Monreale, A., Giannotti, F., Pedreschi, D., Ruggieri, S., & Turini, F. (2019). Factual and counterfactual explanations for black box decision making. *IEEE Intelligent Systems*, *34*(6), 14-23.

[10] Hendricks, L. A., Hu, R., Darrell, T., & Akata, Z. (2018). Generating counterfactual explanations with natural language. *arXiv preprint arXiv:1806.09809*.

[11] Hilton, D. J., & John, L. M. (2007). The course of events: counterfactuals, causal sequences, and explanation. In *The psychology of counterfactual thinking* (pp. 56-72). Routledge.

[12] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in science & engineering*, *9*(3), 90-95.

[13] ImageNet DET_test_dataset , http://image-net.org/image/ILSVRC2017/ILSVRC2017_DET_test_new.tar.gz, Accessed: 2021-01-15.

[14] ImageNet VGG16 Model with Keras, https://slundberg.github.io/shap/notebooks/ImageNet%20VGG16%20Model%20with%20Keras.html, Accessed: 2021-01-17.

[15] Kuhn, R., & Kacker, R. (2019). An application of combinatorial methods for explainability in artificial intelligence and machine learning (draft) (pp. 7-7). National Institute of Standards and Technology.

[16] Kuhn, D. R., Kacker, R. N., Lei, Y., & Simos, D. E. (2020, October). Combinatorial Methods for Explainable AI. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 167-170). IEEE.

[17] Lundberg, S. M., & Lee, S. I. (2017). A unified approach to interpreting model predictions. In *Advances in neural information processing systems* (pp. 4765-4774).

[18] Module:Segmentation – skimage v0.19.0 dev docs, https://scikit-image.org/docs/dev/api/skimage.segmentation.html#skimage.segmentation.slic, Accessed: 2021-02-22.

[19] Molnar, C. (2020). *Interpretable Machine Learning*. Lulu. Com

[20] Mothilal, R. K., Sharma, A., & Tan, C. (2020, January). Explaining machine learning classifiers through diverse counterfactual explanations. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*(pp. 607-617).

[21] Ribeiro, M. T., Singh, S., & Guestrin, C. (2016, August). " Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 1135-1144).

[22] Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). Model-agnostic interpretability of machine learning. *arXiv preprint arXiv:1606.05386*.

[23] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... & Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision*, *115*(3), 211-252.

[24] scikit-image/slic_superpixels, https://github.com/scikit-image/scikit-image/blob/main/skimage/segmentation/slic_superpixels.py, Accessed: 2021-02-22

[25] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[26] Sun, Y., Chockler, H., Huang, X., & Kroening, D. (2020, August). Explaining Image Classifiers using Statistical Fault Localization. In *European Conference on Computer Vision*(pp. 391-406). Springer, Cham.

[27] Vermeire, T., & Martens, D. (2020). Explainable Image Classification with Evidence Counterfactual. *arXiv preprint arXiv:2004.07511*.

[28] XAI-Tool, https://github.com/cjaganmohan/XAI-Tool, Accessed: 2021-01-20

[29] XAI-Tool-Results-Dropbox, https://tinyurl.com/y3sc6qoy, Accessed: 2021-01-20.

[30] 2015, Open Source Computer Vision Library, https://github.com/opencv/opencv, Accessed: 2021-01-20

Chapter 7. Conclusion

In this dissertation, our goal is to address the challenges in testing AI-based software systems. First, we presented a test generation approach that applies combinatorial testing to hyperparameters of an ML algorithm and generates test cases. The results from this experiment helped us to obtain initial understandings about the effectiveness of combinatorial testing on testing hyperparameters of ML algorithms. Second, we presented an empirical study that analyzes the effect of using sampled datasets to speed up the testing of supervised ML algorithms. To the best of our knowledge, this is the first effort to empirically confirm the belief that the size of a test dataset does not significantly impact the coverage metrics (branch and mutation coverage). The result from this empirical study suggests, in most cases, the practitioners can use sampled datasets to accelerate the testing of ML algorithms.

Third, we proposed an approach to generate synthetic images (test data) by systematically combining image transformations to test DNN models used in autonomous driving systems. The results indicate that the synthetic images generated using our approach could detect a significant number of inconsistent behaviors in DNN models. Generating data to test AI systems, particularly for image-based AI systems such as autonomous driving systems, is an expensive and time-consuming process. Also, exhaustive testing is rarely feasible. Therefore, practitioners can adopt the combinatorial testing-based approach to generate test data to test DNNs effectively. Lastly, we presented an XAI approach to produce explanations for decisions made by ML models. The proposed approach showcases that software fault localization techniques can be successfully adopted to derive counterfactual explanations for ML models. As most pre-trained ML models (especially DNNs) are black-box in nature, in the case of misclassification by an ML model, counterfactual explanations (derived using our approach) can help practitioners reason the ML

model's output. In other words, counterfactual explanations can assist the practitioners in model debugging activities.

The work presented in this dissertation could be extended along in several directions. **Testing DNN models using Combinatorial Testing:** 1). A natural progression of this work is to generate synthetic images by systematically combining a). several inclement weather conditions, and b). object segmentation of input images. 2). In addition to this, further studies are required to understand better the impact of synthetic images (generated using t-way testing) on various DNN coverage metrics. 3). Another possible area of future research would be to investigate how to adopt combinatorial testing in the ML model maintenance phase, 4). More broadly, it would be interesting to investigate the possibility of extending the proposed test generation approach to test DNN models used in other domains such as Natural Language Processing and Medical Imaging. **Explainable AI (XAI):** 5). Future studies should extend the XAI approach (presented in this dissertation) to generate explanations for ML models trained with tabular datasets. 6). Furthermore, it would be interesting to investigate how to use the counterfactual explanations to identify the root causes of bugs in DNN models.

BIOGRAPHICAL STATEMENT

Jaganmohan Chandrasekaran was born in Tamil Nadu, India. He received his Bachelor of Technology in Information Technology from Anna University, India. He worked in the industry for three years before joining The University of Texas at Arlington for graduate studies in Computer Science. At the University of Texas at Arlington, he first earned his Master's degree in 2015 and later a Ph.D. in 2021. He served as a Graduate Teaching Assistant in the Department of Computer Science and Engineering at the University of Texas at Arlington from 2015 till 2020. He is a recipient of a STEM fellowship from 2015 – 2021 and a Dissertation Fellowship recipient in 2021. His research interests are in the intersection of Software Engineering and Artificial Intelligence, focusing on the reliability and trustworthiness of AI-based software systems. He wishes to continue working on these areas after the completion of his doctoral program.