Black Box Fairness Testing of Machine Learning Models

Aniya Aggarwal
aniyaagg@in.ibm.com
IBM Research AI
India

Pranay Lohia
plohia07@in.ibm.com
IBM Research AI
India

Seema Nagar
senagar3@in.ibm.com
IBM Research AI
India

Kuntal Dey
kuntadey@in.ibm.com
IBM Research AI
India

Diptikalyan Saha
diptsaha@in.ibm.com
IBM Research AI
India

ABSTRACT

Any given AI system cannot be accepted unless its trustworthiness is proven. An important characteristic of a trustworthy AI system is the absence of algorithmic bias. "Individual discrimination" exists when a given individual different from another only in "protected attributes" (e.g., age, gender, race, *etc.*) receives a different decision outcome from a given machine learning (ML) model as compared to the other individual. The current work addresses the problem of detecting the presence of individual discrimination in given ML models. Detection of individual discrimination is test-intensive in a black-box setting, which is not feasible for non-trivial systems. We propose a methodology for auto-generation of test inputs, for the task of detecting individual discrimination. Our approach combines two well-established techniques - symbolic execution and local explainability for effective test case generation. We empirically show that our approach to generate test cases is very effective as compared to the best-known benchmark systems that we examine.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Individual Discrimination, Fairness Testing, Symbolic Execution, Local Explainability

ACM Reference Format:
Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black Box Fairness Testing of Machine Learning Models. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338937

1 INTRODUCTION

Model Bias. This decade marks the resurgence of Artificial Intelligence (AI) where AI Models have started taking crucial decisions in a lot of systems - from hiring decisions, approving loans, etc. to design driver-less cars. Therefore, dependability of AI models is of utmost importance to ensure wide acceptance of the AI systems. One of the important aspects of a trusted AI system is to ensure that its decisions are fair. Bias may be inherent in a decision-making system in multiple ways. It can exist in the form of group discrimination [6] where two different groups (e.g., based on "protected attributes" such as gender/race) get a varied decision or, through individual discrimination [7] which discriminates between two samples. Note that, discrimination-aware systems need to be trained to avoid discriminating against sensitive characteristic features, that are termed as "protected attributes." Protected attributes are application specific. Features such as age, gender, ethnicity, *etc.* are a few frequent examples of what many applications practically treat as protected attributes [7].

Individual discrimination. In this paper, we address the problem of detecting individual discrimination in machine learning models. The definition of individual fairness/bias that we use in this paper is a simplified and non-probabilistic form of counterfactual fairness [12], which also fits into Dwork's framework of individual fairness [5]. As stated in this work, a system is said to be fair if for any two valid inputs which differ only in the protected attribute are always assigned the same class (and bias is said to exist if for some pair of valid inputs, it yields different classification). Such cases of bias have been previously noticed in models such as [7] and caused derogatory consequences to the model generator. Therefore, detection of such cases is of utmost importance. Note that, removal of such bias cannot be done by removing the protected attributes from the training data as the individual discrimination may still exist due to the possible co-relations between protected and non-protected attributes, just like the case of race (protected) and zip-code (non-protected) in `Adult census income` dataset. [1] This is an instance of indirect discrimination for which individual discrimination testing is still required for co-related non-protected attributes. The challenge is, therefore, to evaluate and find that for which all values of non-protected and protected attributes, the model demonstrates such an individual discrimination behavior.

Existing Techniques and their drawbacks. Measuring individual discrimination requires an exhaustive testing, which is infeasible for a non-trivial system. The existing techniques, such as THEMIS [7], AEQUITAS [23], generate a test suite to determine if and how much individual discrimination is present in the model. THEMIS selects random values from the domain for all attributes to determine if the system discriminates amongst the individuals. The AEQUITAS generates test cases in two phases. The first phase generates test cases by performing random sampling on the input space. The second phase starts by taking every discriminatory input generated in the first phase as an input and perturbing it to generate further more test cases. Both techniques aim to generate more discriminatory inputs. Even though these two aforementioned techniques are applicable to any black-box system, our experiments demonstrate that they miss many such combinations of non-protected attribute

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338937

[1] https://archive.ics.uci.edu/ml/datasets/bank+marketing

values for which the individual discrimination may exist. We also look to cover more diverse paths of the model to generate more test inputs.

Our approach. Our aim is to perform systematic searching of feature space to cover more space without much redundancy. There exist symbolic evaluation [2, 9, 17] based techniques to automatically generate test inputs by *systematically exploring* different execution paths in the program. Such methods avoid generation of multiple inputs which tend to explore the same program path. Such techniques are essentially white-box and leverage the capabilities of constraint solvers [4] to create test inputs automatically. Symbolic execution starts with a random input and analyzes the path to generate a set of path constraints (i.e. conditions on the input attributes) and iteratively toggles (or negates) the constraints in the path to generate a set of new path constraints. It then solves the resultant path constraints using a constraint solver to generate a new input which can possibly take the control to the new path as explained using an example in Section 2. Our idea is to use such dynamic symbolic execution to generate test inputs which can potentially lead to uncovering individual discrimination in ML models. However, existing such techniques have been used to generate inputs for procedural programs which are interpretable. Our main challenge is to apply such technique for non-interpretable models which cannot be executed symbolically. Note that, similar to THEMIS, our goal is to build a scalable black-box solution, which can be applied efficiently on varied models. Black-box analysis will enable AI testing service where a third party machine learning model in the form of an access API can be given as an input along with some payload/training data.

Challenges. There exist a few works which try to use symbolic evaluation-based techniques for non-interpretable models such as deep neural networks, although they do not address the problem of finding individual discrimination that may exist in the model. Such techniques are essentially white-box and try to approximate the functions (ReLu/Sigmoid) that exist in the network. Therefore, they are catered towards a specific kind of networks and are not generalized. Other test-case generation techniques [18, 19] use coverage criteria (like neuron coverage, sign-coverage, etc.) which are structure dependent and hence, such techniques suffer from scalability.

Solution overview. In this paper, our key idea is to use the local interpretable model as the path in the symbolic execution. A local explainer such as LIME [15] can produce a decision tree path corresponding to an input in a model agnostic way. The decisions in the decision tree path are then toggled to generate a new set of constraints. Next, we list several advantages and salient features of our approach.

- *Constraints.* It is possible to use an off-the-shelf local explainer to generate a linear approximation to the path. The linear constraints obtained from one such explainer can be used for symbolic evaluation which won't require any specialized constraint solver [18].
- *Data-driven.* Our algorithm can take advantage of the presence of data which can be used as the seed data to start the search.
- *Global and Local Search.* Once an individual discrimination is found, we perform a local search to uncover many input combinations which can uncover more discrimination. Otherwise, we perform a global search using symbolic execution to cover different paths in the model.
- *Optimizations.* The local explainer presents confidence associated with the predicates. Our algorithm performs the selection of constraints for toggling based on their confidence scores.
- *Scalability.* Our algorithm systematically traverses paths in the feature space by toggling feature related constraints. This makes it scalable, unlike other techniques [18] which consider structure-based coverage criteria.

Contributions. Our contributions are as listed next.

- We present a novel technique to find individual discrimination in the model.
- We develop a novel combination of dynamic symbolic execution and local explanation to generate test cases for non-interpretable models. We believe that the use of local explainer will open up many avenues for path-based analysis of black-box AI models.
- We demonstrate the effectiveness of our technique on several open-source classification models having known biases. We empirically compare our technique with the existing algorithms i.e. THEMIS, AEQUITAS and demonstrate the performance improvement delivered by our approach over these prior works.

Outline. Section 2 presents the required background on dynamic symbolic execution and local explainability. In Section 3, we present our solution while concentrating on the various challenges that we have faced to successfully combine the idea of symbolic execution with the local explanation. Further, in Section 4, we present our experimental setup and the results. Next, we discuss the related prior work in Section 5, while Section 6 mentions a brief summary along with the possible future extensions of this work.

2 BACKGROUND

2.1 Notation and Individual Discrimination

Let us consider a dataset $\mathcal{D}$ with the set of attributes defined as $A = \{A_1, A_2, \ldots, A_n\}$ where $P \subset A$ denotes its set of protected attributes. We use x, x' to represent two data instances, where $x, x' \in \mathcal{D}$. Each data instance is an n-tuple $x = (x_i, \ldots, x_n)$ where x_i denotes the value for A_i in data instance x. The domain of attribute A_i is denoted as $Dom(A_i)$. Say, a model M is trained on the training data TD, then $M(x)$ denotes the output of this model on an input x. Formally, an individual bias instance is a pair (x, x') of input samples such that: $x_i = x'_i$ for all $A_i \notin P$ and $\exists j, x_j \neq x'_j$, where $A_j \in P$ and $M(x) \neq M(x')$.

Note that we use a strict notion of equality for all non-protected attributes which does not consider various issues such as equality in case of continuous attributes, possible co-relation between attributes, etc. However, the focus of the paper is to perform systematic test case generation with respect to a definition of individual fairness and therefore, handling the above issues are considered as possible future extensions.

2.2 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) [2, 9, 17] performs automated test generation by instrumenting and running a program while collecting execution path constraints for different inputs. It systematically toggles the predicates present in a set of path constraints

Algorithm 1: Generalized Dynamic Symbolic Execution

```
1  count = 0;
2  inputs = seed_test_inputs()
3  priorityQ q = empty; q.enqueueAll(inputs,0)
4  while count<limit && !q.isEmpty() do
5      t = q.dequeue()
6      check_for_error_condition(t)
7      Path p = getPath(t)
8      prefix_pred = true
9      foreach predicate c in order from top of path do
10         path_constraint = prefix_pred $\wedge$ toggle (c)
11         if !visited_path.contains(path_predicate) then
12             visited_path.add(path_predicate)
13             input= solve(path_constraint)
14             r = rank(path_constraint)
15             q.enqueue(input,r)
16         end
17         prefix_pred = prefix_pred $\wedge$ c
18     end
19     count++
20 end
```

to generate a new set of path constraints whose solution generates new inputs to further explore a new path. Note that this technique caters for path coverage and hence, does not generate any redundant input. The path coverage criteria states that the test cases are executed in such a way that every possible path is executed at least once. It is further to be noted that the path coverage criteria are stronger than branch/decision coverage and statement coverage.

Next, we formally present a generalized version of the above algorithm in Algorithm 1 which can be used as a framework to generate test cases for AI models. We further discuss the changes in this generalized version as compared to the algorithm for dynamic symbolic execution used in DART [9].

The first change relates to the inputs the algorithm starts with. Instead of starting from a random input, the algorithm finds one or more seed inputs to start with (Line 2). Seeds can also be generated at random. The second change (Lines 3, 14, 15) is related to the abstraction of the ranking strategy of selecting which test inputs to execute next. Please note that the `rank` function now determines which input should be processed next. The third change is an addition of the check to detect if a path is already traversed or not (Line 11). Such checks are not required in symbolic execution for programs as the selection of predicates for toggling ensures that an already traversed path will not be traversed again. However, in an environment where path generation does not guarantee preservation of predicates in the path, this is a necessary check to avoid generating redundant inputs.

The generation of constraints (Lines 9-17) corresponding to the generalized algorithm is illustrated in Figure 1. Note that the other variables, not present in the constraint, can take any value from its possible domain.

2.3 Local Explainability

Local Interpretable Model-agnostic Explanations (LIME) [15] consists of explanation techniques that explain any prediction of any classifier or regressor in an interpretable and faithful manner. It is able to do that by approximating it as an interpretable model locally around the prediction. It generates explanation in the form of interpretable models, such as linear models, decision trees, or falling rule lists, which can be easily comprehended by the user with visual or textual artifacts. Given an input data instance and its output as generated by the classifier, LIME generates data points in the vicinity of the instance by perturbation of the input data point and generates the output. Using such input and output, it learns an interpretable model by maximizing local-fidelity and interpretability.

We use LIME to explain a prediction instance for a model and generate a decision tree as interpretable model for a prediction.

3 ALGORITHM

In this section, we discuss our overall solution approach spread across the subsequent three subsections. The first sub-section discusses the goals of our test case generation technique under different conditions. We divide our algorithm into two different kinds of search algorithms called global search and local search. The next two sub-sections devout on these respective searches.

3.1 Problem Formulation

Below are the two optimization criteria that we want to achieve through the devised test case generation technique.

Effective Test Case Generation: Given a model M, a set of domain constraints C and protected attribute set P, the aim is to generate test cases to maximize the ratio of $|Succ|/|Gen|$, where Gen is the set of non-protected attribute value combinations generated by the algorithm and $Succ \subseteq Gen$ that leads to discrimination i.e. each instance in $Succ$ yields at least one different decision for different combinations of protected attribute values.

Here are a few salient points about this criteria.

- *Test case*: Each test case is not considered as the collection of the values of all the attributes, but only non-protected attributes. This ensures that multiple discriminatory test cases are not counted for the same combination of non-protected attribute values.
- *Domain constraints*: We assume that applying domain constraints C will filter out unreal test cases.
- *Order of generation and discrimination test*: The optimization criteria does not specify if all the test cases are generated at once or whether check for discrimination and generation can go hand-in-hand. This way the test case generation can be dependent on discrimination checking as well.

In the software testing domain, there exists a number of predefined coverage criteria. Many such coverage criteria have also been defined in recent works on machine learning [19]. Next, we define the path coverage criteria such that it is applicable for varied types of models.

Coverage criteria: Note that defining path coverage criteria for *any* black-box model is not straightforward. It is possible to define

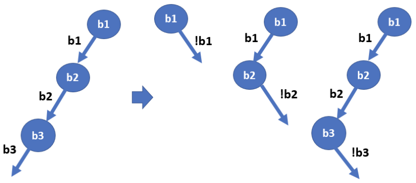

Figure 1: Generation of path constraints

Algorithm 2: Individual Discrimination Checker

```
1  function generate_test_cases($seed, model, cluster\_count, limit, Rank1,$
     $Rank2, Rank3$):
2      count = 0;
3      priorityQ q = seed_test_input($seed, cluster\_count, limit, Rank1$)
4      while count<limit && !q.isEmpty() do
5          $t$ = q.dequeue()
6          found = check_for_error_condition($t$)
7          Path p = get_path($model, t$)
8          if found then
9              // local search
10             foreach predicate $c$ in order from top to bottom of $p$ do
11                 path_constraint = p.constraint
12                 If $c$ is of protected attribute then continue
13                 path_constraint.remove(c) path_constraint.add(not(c))
14                 if !visited_path.contains(path_constraint) then
15                     visited_path.add(path_constraint);
16                     input= solve(path_constraint,t)
17                     r = average_confidence(path_constraints)
18                     q.enqueue(input,Rank2-r)
19                 end
20             end
21         end
22         // global search
23         prefix_pred = true
24         foreach predicate $c$ in order from top to bottom of $p$ do
25             If $c$ is a protected attribute then continue
26             If $c.confidence < T1$ then break
27             path_constraint = prefix_pred ∧ not (c)
28             if !visited_path.contains(path_constraint) then
29                 visited_path.add(path_constraint);
30                 input= solve(path_constraint)
31                 r = average_confidence(path_constraints)
32                 q.enqueue(input,Rank3+r)
33             end
34             prefix_pred = prefix_pred ∧ c
35         end
36         count++
37     end
38     return
39 function get_path($model, input$):
40     Set<In,Out> inout = LIME_localexpl(model,input)
41     return genDecisionTree(inout);
42 function seed_test_input($data, cluster\_count, limit, Rank1$):
43     i = 0
44     priorityQ q = empty
45     clusters=KMeans(data, cluster_count)
46     while $i <$ max(size(clusters)) do
47         if q.size() == limit then
48             break
49         end
50         foreach cluster ∈ clusters do
51             if $i \geq$ size(cluster) then
52                 continue
53             end
54             row = cluster.get(i)
55             q.enqueue(row, Rank1)
56         end
57         i++
58     end
59     return (q)
60 function check_for_error_condition($t$):
61     class = M(t)
62     foreach $\langle val_0, \ldots, val_n \rangle | \forall A_i \in P, val_i \in Dom(A_i)$ do
63         // try combination of protected attribute values
64         tnew=Replace value of $A_i$ in $t$ with $val_i$
65         class1 = M(tnew)
66         if class1 != class then
67             return Individual Discrimination Found
68         end
69     end
70     return Individual Discrimination Not Found
```

path for different types of models based on their operational characteristics. For instance, it is possible to define a path in a neural network based on the activation of neurons (similar to the branch coverage defined in [19]), and decision paths in a decision tree classifier.

We define the coverage criteria as follows. Given a classification model M and a set of test cases T, we define the coverage of T as the number of decision regions of M executed by T.

In this paper we use a decision tree classifier to approximate the behavior of the model M. We generate highly accurate decision tree model to approximate the decision regions of M.

The aim of our test case generation technique is to maximize both the path coverage and the individual discrimination detection. In practice, there is always a limit to the automatic test case generation process within which maximizing both the objectives needs to be done. In our case, we consider two such possible limits: 1) the number of generated test cases, and 2) the time taken to generate.

In the subsequent subsections, we present our algorithm that intends to maximize path coverage and effectively detects discrimination, and the way to combine them both.

3.2 Maximizing Path Coverage

Path coverage maximization is carried out by leveraging the capabilities of symbolic execution algorithm which is catered towards the systematic exploration of different execution paths. In this subsection, we describe the various functions that are kept undefined in Algorithm 1. The final algorithm is presented in Algorithm 2. Maximizing the path coverage is done in the global search module as mentioned in our final algorithm presented in Algorithm 2.

Path Creation. Let us start with the `get_path` method. As mentioned in Section 1, our key idea is to use the set of input, output as generated from a local explainer - LIME. Our algorithm learns the decision tree using the input and output instead of the linear regression classifier as in the original implementation of LIME (refer Line 39 in Algorithm 2).

With above definition of `get_path` method, the algorithm works in the same way as the symbolic execution algorithm, which can negate the conditions present in the decision tree path to generate new set of constraints that can then be solved using a constraints solver to generate further new inputs that can traverse other paths.

There are three major challenges associated with the straightforward application of the idea of symbolic execution and local model approximating a path. The first two arise due to the inherent approximation present in the local model, while symbolic execution is the reason for the third one.

- *Approximation*: The decision tree path approximates the actual execution path in terms of interpretable features. Because of such an approximation, duplicates of the actual program path can be generated.
- *Confidence*: Decision tree path has a confidence score associated with all its comprising predicates (which was not the case for a program path). Therefore, the challenge lies in devising a way to use this confidence score for better exploration of the paths.
- Symbolic execution in program testing suffers from path explosion problem [8], especially in the depth-first-search way. It can keep on exploring paths in the depth of the program tree, without exploring paths in other parts of the program. Researchers have explored various techniques to address

this problem - applying demand driven or directed technique which generates test cases towards a particular location in the program, and compositional techniques which try to analyze various functional modules separately before combining them to generate longer paths in the whole program. All these techniques exploit the structure of the program under test.

Addressing Path Explosion. To resolve the path explosion problem, such that the path generation does not concentrate only on a few localized portion of the entire space, we exploit the distribution present in the data (training or testing), if available. Each data instance can be a good starting point of the symbolic search. Therefore, we add the data as the seed test data, as shown in method at Line 42 given in Algorithm 2.

However, the order of data instances become important when a search limit is imposed due to which execution of all the data instances is not possible. Therefore, to increase the diversity in search, we cluster the data and process seed inputs from each cluster in a round-robin fashion.

Addressing Local Model Approximation. We use an off-the-shelf local explainer (LIME) to fetch the interpretable local model. The perturbation used in such an explainer is outside the scope of the present algorithm, therefore, it is not possible to reduce the error caused by the local approximation. Note that, the approximation can lead to production of test inputs which do not uncover any new path in the model. However, by introducing the data instances in the seed, our algorithm addresses the above issue. Our algorithm further ensures that the seed inputs get more priority than the inputs generated by the constraint solver during symbolic execution. This ensures high degree of varied path coverage.

Ranking based on Confidence. Automated test case generation procedure typically runs within a limit. It is therefore important to generate non-redundant test inputs and cover as much as path covered by the generated test inputs within the limit. For the test cases generated by the constraint solver, we use a ranking scheme based on the confidence of predicates in the decision tree to select which test input to execute next. The confidence of a path is determined by taking an average of the confidence of all its comprising predicates (Line 31, Algorithm 2). Therefore, when the algorithm selects a predicate c for negating, it considers the rank as the average confidence of the predicates in the prefix of the path leading to and including c.

Confidence Threshold. The lesser the confidence of a predicate in the decision tree path, the lesser is the chance of generating varied inputs. Therefore, to remove unnecessary generation of inputs through symbolic evaluation, our algorithm employs a threshold on the confidence of the predicate for selection (Line 26, Algorithm 2). The value of the threshold is obtained through experimentation.

3.3 Maximizing Effectiveness of Discrimination Detection

In this subsection, we discuss a few more changes that we have employed on the generic algorithm in order to maximize the detection of individual discrimination.

Checking Individual Discrimination: To start with, let us consider the case of checking individual discrimination which is carried out in the method `check_for_error_condition`, as presented in Algorithm 2. The algorithm performs the check as per the definition of individual discrimination. A test case is said to be individually discriminatory if keeping the values of its set of non-protected attributes same, but changing the values of its protected attributes set by trying every possible combination, yields different class labels.

Local Search: The symbolic execution as discussed in earlier sections, tries to find test inputs to maximize the path coverage. We call such a symbolic search strategy the *global search.* Some of the test inputs generated through seed data or symbolic execution will be discriminatory in nature. To increase the likelihood of discriminatory test cases, we exploit the fact that we can execute test cases and check whether they are discriminatory and then, depending on that, generate more test cases.

Once a discriminatory test case, say t is found, we try to generate further more test inputs which may lead to individual discrimination. The key idea is *to negate the non-protected attribute constraints of t's decision tree to generate more test inputs.* By toggling one constraint related to non-protected attribute and generating an input solving the resultant constraint, the algorithm tries to explore the neighborhood of discriminatory path p. This form of symbolic execution is what we call as *local search* as it tends to search the locality of discriminatory test cases.

The reason why this works is due to the inherent adversarial robustness property of a machine learning model which demonstrates that a small perturbation of an input can result in changing the classifier decision [20].

Sticky Solutions. The aim of the local and the global search is to traverse as many paths as possible. The local search concentrates on exploring the paths in the vicinity of the discriminatory paths i.e. the paths that are generated from discriminatory inputs. Therefore, we only get one solution for the constraint. However, to cater to the possible approximation caused by the local linear model, we use the constraint solver's solution that is close to the solution of the previous constraint (related to discriminatory input). We call such a solution as a *sticky solution.* Because of the stickiness, if we negate one predicate, then for the remaining predicate, it tends to take the same values as in its previous solution. In Section 4, we demonstrate that sticky solution provides a huge performance improvement over the existing works.

The algorithm for test case generation to detect individual discrimination is presented in method `generate_test_cases` given at Line 1 Algorithm 2. Lines 10-21 describe the local search whereas Lines 23-34 describe the global search. There are two major differences between the implementations of the local and global search. The first difference is that no threshold based constraint selection is done in local search. In contrast, during global search, only the high confidence constraints are chosen for toggling. The reason behind this strategy is to increase the chances of diverse coverage of paths. The second difference is that, in global search, no constraint exists for suffix of the path, whereas in local search, all constraints, except the selected low confidence one to toggle, remain as it is.

Ordering Local and Global Search. In the consolidated Algorithm 2, three reference ranks, namely *Rank*1, *Rank*2 and *Rank*3 are presented, one each for seed input, local search, and global search respectively. These ranks are set in such a way that the highest priority is given to local search, followed by the seed input, which is further followed by the global search, based on their ability to uncover discrimination causing inputs (see Lines 3, 18, 32 of Algorithm 2).

In the next section, we experimentally show the effectiveness of various optimizations described in this section along with comparisons with the existing works.

4 EXPERIMENTAL EVALUATION

4.1 Setup

4.1.1 Benchmark Characteristics. We have conducted our experiments on eight open-source fairness benchmarks from various sources as listed in Table 1.

4.1.2 Configurations. Our code is written in Python and executed in Python 2.7.12. All the experiments are performed in a machine running Ubuntu 16.04, having 16GB RAM, 2.4Ghz CPU running Intel Core i5. We have used LIME [15] for local explainability. We have used *K-means* with cluster size = 4 to cluster the input seed data. Since our use case requires generation of more test cases in lesser time, *K-means* being one of the simplest and fastest clustering algorithms, proves to be a reasonable choice. The fact that the data sets used to run our experiments have either two or four true class labels, drives the logical assumption to set the cluster count as 4. This was further validated using a scatter plot, shown in Figure 2, which clearly depicts four different clusters in the seed data.

For each benchmark, we have generated a model using *Logistic Regression* with its default configuration as provided in *scikit-learn*. The model is configured similar to the one used in THEMIS [7] in order to ensure fair comparison. The models are highly precise (accuracy > 95%) to avoid overfitting. The threshold $T1$, proposed earlier in Algorithm 2, is set to $T1 = 0.3$. The justification for choosing this value of threshold has been experimentally validated later in this section.

4.2 Experiment Goals

By conducting our set of experiments, we broadly try to achieve two goals, as mentioned below.

Table 1: Benchmark Characteristics

Benchmark	Size	Features Count	Domain Size
German Credit Data [1]	1000	21	6.32×10^{17}
Adult census income [2]	32561	12	1.95×10^{16}
Bank marketing [3]	45211	16	3.30×10^{24}
US Executions [4]	1437	10	1.40×10^{8}
Fraud Detection [5]	1100	9	4.01×10^{9}
Raw Car Rentals [6]	486	7	1.15×10^{3}
credit data [1](modified)	600	20	1.74×10^{15}
census data [2](modified)	15360	13	1.74×10^{18}

1 https://archive.ics.uci.edu/ml/datasets/adult
2 https://archive.ics.uci.edu/ml/datasets/bank+marketing
3 https://data.world/markmarkoh/executions-since-1977
4 https://www.kaggle.com/c/frauddetection/data
5 https://www.yelp.com/biz/raw-car-rentals-sacramento

- *Comparison with the existing works* - The intention is to gauge how well does our algorithm perform as compared to the existing works in the field of test case generation to find individual discrimination in models. We compare our approach with two existing systems: (a) THEMIS [7], which checks the individual discrimination by random test case generation, and (b) AEQUITAS [23], which performs both random global search and perturbation-based local search. We have used *success score* (i.e. Succ/Gen) as an appropriate metric for evaluation, where *Succ* denotes the subset of the generated test cases i.e. *Gen* which results in individual discrimination. Please note that the same metric as ours has also been used in other related works such as THEMIS and AEQUITAS. Furthermore, precision and recall as evaluation metrics are more appropriate when we have fixed gold standard data. A detailed theoretic comparison with both the approaches is presented in the forthcoming Section 5. In this paper, we refer our algorithm as SG which stands for Symbolic Generation, in all our subsequent discussions.
- *Effect of algorithmic features* - The motive is to find out how well does each algorithmic feature, such as symbolic execution in local and global search, presence of data, etc., contribute towards finding individual discrimination.

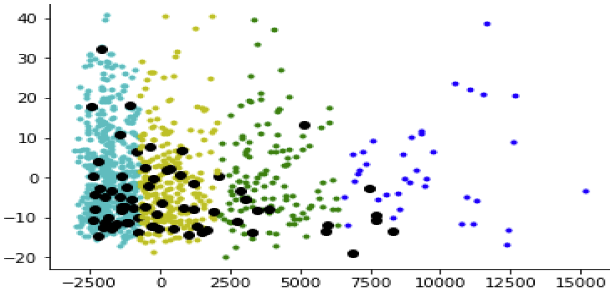

Figure 2: Scatter plot for Clustered Seed Data and Discriminatory Inputs (Black) for German (age) data

4.3 Comparison with Related Work

4.3.1 Comparison with THEMIS. We have fetched THEMIS code from their GitHub repository.[2] On carefully analyzing their code, we figured out an unintended behavior in the open-sourced code. THEMIS actually generates duplicate test cases and their reported experimental statistics also contains these duplicates. This is one of the problems posed by random test case generation as it can produce duplicate test cases. We have changed THEMIS's code to remove duplicates for our experimental evaluations.

Table 2 depicts the results of the comparison with THEMIS. It can be inferred from the results that our algorithm performs better than THEMIS, except in the case of one benchmark i.e. Fraud Detection. THEMIS has an average success score (#Succ/#Gen) of 6.4% but, for our algorithm it is 34.8%. It is evident that **across 12 benchmarks, our algorithm generates 6 times more successful** (i.e. the ones that resulted in discrimination) test cases than THEMIS. It is further to be noted that the maximum percentage of discriminatory test

[2] https://github.com/LASER-UMASS/Themis

Table 2: Comparison of SG with THEMIS

Bench.	Prot. Attr.	THEMIS		SG	
		#Gen	#Succ	#Gen	#Succ
German Credit	gender	999	166	1000	640
German Credit	age	999	90	1000	485
Adult income	race	999	70	1000	295
Adult income	sex	990	1	1000	858
Fraud Detection	age	999	3	1000	0
Car Rentals	Gender	680	198	1000	783
credit	i/gender	598	44	1000	267
census	h/race	999	57	1000	970
census	i/sex	999	7	1000	282
Bank Marketing	age	999	0	1000	1
US Executions	Race	999	2	1000	6
US Executions	Sex	999	8	1000	31

Table 3: Contribution of different features

Bench.	Data		Local Symb.		Global symbolic	
	Gen	Succ	Gen	Succ	Gen	Succ
German Credit(gender)	25	2	975	638	0	0
German Credit(age)	25	2	975	483	0	0
Adult Income(race)	37	4	963	291	0	0
Adult Income(sex)	35	6	965	852	0	0
Fraud Detection	1000	0	0	0	0	0
Car Rentals	2	2	998	781	0	0
credit	22	2	978	265	0	0
census(race)	1	1	999	969	0	0
census(sex)	6	1	994	281	0	0
Bank Marketing	983	1	17	0	0	0
US Executions(Race)	877	0	22	4	101	2
US Executions(Sex)	877	0	74	29	49	2

cases as generated by SG is 97%, which is also very high as compared to THEMIS’s 29%.

In Table 3, we report the contribution of different test case generation features, such as seed data, global symbolic search and local symbolic search, towards the aforementioned success. The *data* refers to the case where test instance itself is the data point available in the input seed data. *Global and Local symbolic search* refer to the cases where test cases are generated by employing symbolic execution in the global search and local search, respectively. Please note that the contribution of each of these searches is dependent on the previously executed search. Table 3 does not aim to compare the effectiveness of these individual searches, but records whether a test case is generated in the data/local/global symbolic search phase, thus drilling down on results of Table 2.

The results show the effect of our relative ranking strategy which specifies the order of preferences for local symbolic, seed data, and global symbolic in the decreasing order. Please note that on an average, the success percentage for data and local symbolic execution are 22.4% and 47.4%, respectively.

We have performed an another experiment where we try to make the same comparison with THEMIS, but by varying the time limit instead of considering a fixed limit of test cases to be generated. Essentially, we conducted this test to check that within a given time limit, whether SG performs better than THEMIS or not. Considering that our algorithm requires local model generation and running constraint solver, this experiment also takes into account the time taken by such components in our algorithm. This result also depicts the time required for test case generation by our algorithm. We experimented with *German Credit* dataset with ’age’ as its protected attribute and run both THEMIS and SG for 2, 000 secs. Interestingly, at any given instant, the number of test case generated by SG is more than that of THEMIS. The reason is that random sample generation takes a lot of time as it still conforms to the constraints of the domain, whereas, SG takes very little time to generate samples from seed and LIME/constraint solver-based local and global test case generation is also quite fast. Most importantly, within the same limit, the number of discriminatory test cases for THEMIS is much lower as compared to SG, which is well evident by the results in Table 4.

Table 4: Time limit wise comparison between THEMIS and SG for German Credit Dataset

Duration	THEMIS		SG	
(secs)	Gen	Succ	Gen	Succ
10	3	0	12	0
20	4	0	22	0
50	8	0	51	2
100	12	0	89	14
200	33	2	58	41
500	114	5	341	113
1000	282	21	649	232
2000	569	40	1000	359

Table 5: Number of Discriminatory Test Cases generated through Global Search for Adult Dataset with limit 1000

Model Type	AEQUITAS	SG
Decision Tree	6	56
Random Forest	73	93
Multi Layer Perceptron	41	48

4.3.2 Comparison with AEQUITAS. The AEQUITAS [23] algorithm operates in two search phases - global and local. The global phase considers a limit on the count of test cases and generates them by random sampling of the input space. Out of all these generated test cases, a few of them are discriminatory in nature. The local phase then starts by taking each of the discriminatory inputs identified in the global search phase as an input and perturbs it to generate further more test cases. This phase just like the previous global search considers a limit on the number of test cases to be generated. They have applied three different types of perturbation resulting in three different variations of the algorithm.

As we have similar phases in our algorithm too, therefore, we perform phase-by-phase comparison with AEQUITAS by considering the same limit. We have fetched the code for AEQUITAS from their Github[3] repository and run it for comparison.

Global Search Comparison. Table 5 presents a comparison of SG with AEQUITAS in context of their global search strategy. Our global search method uses clustered seed data and symbolic execution, whereas their strategy uses random sampling of the input space. It is evident from the statistics that in general, our algorithm generates more discriminatory inputs othan AEQUITAS.

Local Search Comparison. AEQUITAS perturbs the available discriminatory test input to find even more discriminatory inputs. In contrast, SG’s local search is still symbolic which helps to discover more execution paths near the discriminatory paths. To accurately compare both the local searches, we have run them considering the *same set* of discriminatory inputs and the same models for the Adult dataset. Please note that AEQUITAS open-source code has

[3] https://github.com/sakshiudeshi/Aequitas

Table 6: Number of Discriminatory Test Cases generated through Local Search for Adult Dataset with limit 1000

Model Type	AEQUITAS	SG
Decision Tree	318	694
Random Forest	541	726
Multi Layer Perceptron	403	42

hard-coding for the `Adult` dataset and therefore, we could only perform the comparison experiments on that dataset but for all the three different model types. Table 6 presents the comparison of the number of discriminatory inputs generated by both the approaches within the limit of 1000 test cases. In two out of three cases, SG's local search performs better than that of AEQUITAS's. In case of Multi Layer Perceptron, SG's poor performance is attributed to the error in local model approximation by LIME.

4.4 Path Coverage

We perform an experiment to compare the path coverage of our global search and random data based search. The random data based search has been applied for both THEMIS and AEQUITAS. Therefore, this experiment presents the comparison with the existing related works.

To perform the path coverage, we learn a decision tree model with a precision of 85%-95%, measured using 5-fold cross-validation for each benchmark and map each generated test input to a path of the decision tree model. The results in Table 7 show that on average over all the benchmarks, SG has 2.66 times more path coverage than random data. This result demonstrates that on the path coverage metric, we are superior to other algorithms. Therefore, our algorithm will be able to find discriminatory inputs at various different places in the model. This is important as at one shot we can de-bias multiple parts of the model if we use the test cases for retraining.

4.5 Importance of Seed Data

We conducted two experiments for determining the importance of seed data and related features. In both the experiments we compare the training data with random data as seed data.

In the first experiment (shown in Table 8), we switch off the global and local symbolic execution so that all the test cases are generated from the seed data i.e. the training data. In the second experiment (Table 9), we keep both the symbolic searches. Both the experiments are performed with a limit on the number of test cases generated. The third experiment compares the path coverage of training data vs. the random data.

Limited Number of Test Cases: The results from the first experiments in Table 8 demonstrates the importance of the seed data. The results show that for all benchmarks, the number of discriminatory test cases generated from seed training data is more than that of random data. We see that just by applying training data (without any symbolic search) we get an average improvement of 108% (25% to 12%) in comparison with random data. This implies that in global search, more discriminatory examples can be obtained with training data. This result also demonstrates why our global search method is superior to THEMIS and AEQUITAS to find discriminatory inputs.

Table 7: Path coverage of SG-global search vs Random

Benchmark	Limit	Random	SG-Global
adult	1000	284	610
census	1000	47	413
credit	600	61	116
German (age)	1000	102	187

The second experiment in Table 9 shows the effectiveness of symbolic evaluation even if we start with the random input. For example, in the `credit` data, we see that random data got 310 successful test cases and is less effective than training data which has got 421 successful test cases.

Importance of Clustering of Seed Data. Figure 2 shows the distribution of the seed data and the discriminatory inputs found. It also demonstrates that the discriminatory inputs are scattered across the input space, a proof why random exploration of the input space is not effective in finding discriminatory inputs.

Figure 3 shows the use of diverse seed data ordering got getting individual discrimination. When the number of test cases is limited then we expect that diverse ordering (round-robin) will fetch more discrimination than iterative ordering. It is evident from the Figure 3 that, for most number of test cases, the number of discrimination found is higher for round-robin selection from clusters is more effective than iterative selection. For example, in the execution of 600 test cases, round-robin selection found 45 discriminatory inputs, compared to 35 found by iterative selection.

Note that, even on using random data as seed, our global symbolic search performs well, as discussed in the next subsection.

Table 8: Training vs Random seed data (no symbolic execution)

Bench.	Training Data		Random	
	Gen	Succ	Gen	Succ
credit	500	56	500	25
German (Age)	1000	70	1000	46
Census (Sex)	500	20	500	5
Car	500	394	500	190

Table 9: Random seed data (with symbolic) (#Succ/#Gen)

Bench.	Random			
	Total	Seed	Local Symbolic	Global Symbolic
credit	310/1000	1/21	309/979	0/0
German (Age)	365/1000	4/49	361/951	0/0
Census (Sex)	195/1000	3/87	192/913	0/0
Car	803/1000	14/21	789/979	0/0

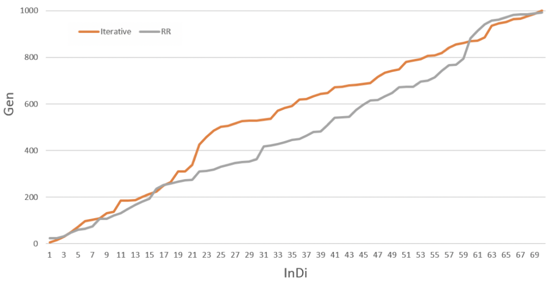

Figure 3: Importance of clustering (German-age)

Table 10: Global Symbolic Search (w/o local) with Random Seed Data (#Succ/#Gen)

Bench.	Total	Seed	Local	Global	Total Random
credit	14/1000	14/773	0/0	0/227	18/1000
German (Age)	446/1000	0/1	0/0	446/999	380/1000
Census (Sex)	16/1000	1/883	0/0	15/117	1/1000
Car	655/1000	1/1	0/0	654/999	626/1000

4.6 Importance of Global Symbolic Search

To determine the effectiveness of global symbolic search we performed an experiment where we removed the local symbolic search to emphasis on the global search part and we assigned the higher priority to the global symbolic search than the seed data. Since the seed data selection may also affects the symbolic search, we used random data as seed data. The result of the experiment is shown in Table 10. We see that in case of `credit`, effectiveness is little lower (14 vs 18) as symbolic search could not find any discriminatory cases generated in the generated 227 test cases. In all the other three cases, global symbolic search alone is able to generate 22% more discriminatory test cases than random search.

Importance of Threshold in the Global Symbolic Search: We experimented with the global search threshold for 4 benchmarks and the result is shown in Figure 4. The result shows that in most cases the lower threshold causes decrease in the ratio of discriminatory inputs. Therefore, we have chosen the threshold to be 0.3, which balances the number of test case generated and the effectiveness of generation.

4.7 Importance of Local Symbolic Search

Based on the previous experiments we notice that local symbolic search has a high percentage (47.4%) of effectiveness. We conduct another experiment to see how we local symbolic search affects the overall execution of the algorithm given a limit on the number of test cases (1,000). The results by switching off the local symbolic search feature is presented in Table 11. We should compare this result with the results in Table 3 for the 4 benchmarks. We see that the average effectiveness drops from 45.4% to 25.2% for these 4 benchmarks by removing the local symbolic search feature. This shows the importance of the local symbolic search technique.

Overall, our experiments demonstrate that local symbolic search uncovers many bias instances after finding the discriminatory input.

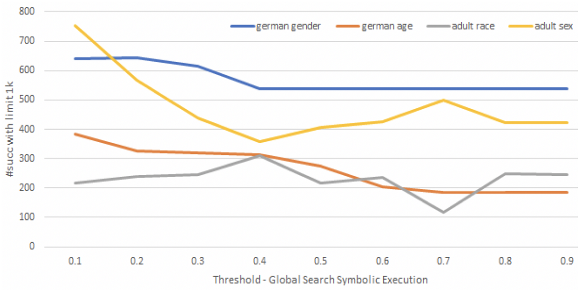

Figure 4: Importance of Threshold

Table 11: Without local symbolic search (#Succ/#Gen)

Bench.	Total	Seed	Local	Global
credit	66/603	66/600	0/0	0/3
German (Age)	70/1000	70/1000	0/0	0/0
Census (Sex)	45/994	45/992	0/0	0/2
Car	231/295	91/114	0/0	140/181

For initial fault finding, seed data works better than the global symbolic search.

4.8 Threats to Validity

Nature of data inputs. Our approach works well on the data sets containing both numeric and categorical attributes, or a mix of them. However, it doesn't handle very high dimensional data, such as image, sound or video, in its current state due to a limitation posed by our local explainer, LIME. Note that, our algorithm is generic to any local explainer. So, LIME can be replaced by any sophisticated explainer, such as Grad-CAM [16] in the image domain, to make it work for the high dimensional data.

Relevance of benchmarks and models used. The benchmarks used to evaluate this work are well-known in the field of fairness and are used in the related works, such as THEMIS [7] and AEQUITAS [23] as well. Further, we have picked the same models to conduct our experiments as the ones used in these works.

Protected attributes. We consider only one protected attribute at a time per benchmark for our experiments. However, considering multiple protected attributes will not hamper the coverage or effectiveness offered by our novel search technique, but will certainly lead to an increase in execution time. This increase is attributed towards the fact that the algorithm in such a case, needs to consider all the possible combinations of their unique values.

5 RELATED WORK

This section discusses the existing works spread across two related spheres - model testing and individual discrimination detection.

AI Model Testing. First, we present the works which are capable to perform symbolic or concolic-based test case generation for AI models. *DeepCheck* [10] uses a white box technique which performs symbolic execution on deep neural networks with the target of generating adversarial images. Another related work [19] performs concolic execution [9] (Concrete and Symbolic) on deep neural networks. Their technique is also white-box with the goal of ensuring coverage of deep neural network by systematic test case generation. They model the network using linear constraints and use a specialized solver to generate test cases. Wicker et al. [24] aim to cover the input space by exhaustive mutation testing that has theoretical guarantees, while in [13, 14, 21] gradient-based search algorithms are applied to solve optimization problems. In addition, Sun et al. [18] leverage the capabilities of linear programming.

All above techniques are white-box as compared to our black-box solution approach. We use an off-the-shelf solver to generate test cases. The aforementioned approaches try to consider test generation to create adversarial input in the image space, while our technique addresses a new problem domain - *trust and ethics.*

Individual Discrimination Detection. THEMIS [7] uses the notion of causality to define individual discrimination. Even though they use a black-box technique, their technique employs random test case generation approach instead of a systematic one. However, they envision the use of systematic test case generation techniques in their paper as future work.

AEQUITAS [23] randomly samples the input space to discover the presence of discriminatory inputs. Then, it searches the neighborhood of these inputs to generate further more inputs. Therefore, their global search technique is based on random search, while the local search is performed by perturbing discriminatory inputs. However, their approach still differs from ours in the context of global search and local search techniques. For global search, they perform random sampling on the input space, while we use clustered seed data and symbolic execution to systematically uncover the first set of discriminatory inputs. The effectiveness of their technique completely depends on where the discriminatory inputs are placed. As we have already demonstrated in Section 4, our global search performs better than random sampling because of the combination of clustered seed data and symbolic execution used in our technique. In the context of local search, the two techniques differ in the way that they perturb. AEQUITAS perturbs the input by a factor of $\delta \in \{-1, +1\}$ (0.5 to start with) and selects attributes with a probability p (which is set as uniform across non-protected attribute at the start). The probability of selection and δ is further updated depending on their three different strategies, namely random, semi-directed, and fully-directed. In contrast, our selection of attributes is based on the confidence of the predicates associated with the attributes and the change (or perturbation) is performed based on the constraints found in the approximated decision tree path. AEQUITAS tends to generate test data points in the vicinity of the discriminatory test cases which may or may not explore new paths, but our algorithm makes use of symbolic execution which results in exploration of new paths near the discriminatory ones. Furthermore, our perturbation scheme does not consider any pre-determined offset like the one used in AEQUITAS.

FairTest [22] uses manually written tests to measure four types of discrimination scores. Their idea is to leverage indirect co-relation behavior existing between attributes (e.g., salary is related to age) to generate test cases. FairML [1] uses an iterative procedure based on an orthogonal projection of input attributes, to enable interpretability of black-box predictive models. Through such an iterative technique, one can quantify the relative dependence of a black-box model on its input attributes. The relative significance of the inputs with respect to a predictive model can then be used to assess the fairness (or discriminatory extent) of such a model.

Despite being black box techniques, none of these existing individual discrimination techniques uses systematic test case generation. To the best of our knowledge, we are the first ones to present a black-box solution approach capable of generating test cases systematically with the intention of detecting individual discrimination in AI models.

6 CONCLUSION AND FUTURE WORK

In this paper, we present a test case generation algorithm to identify individual discrimination in machine learning models. Our approach combines the notion of *symbolic evaluation* which systematically generates test inputs for any program, with *local explanation* that approximates the execution path in the model using a linear and interpretable model. Our technique offers an additional advantage by being black-box in nature. Our search strategy majorly spans across two approaches, namely global and local search. Global search caters for path coverage and helps to discover an initial set of discriminatory inputs. In order to achieve that, we use seed data along with symbolic execution while considering approximation existing in the local model and intelligently using the confidence associated with the path constraints fetched from the local model. Further, local search aims at finding more and more discriminatory inputs. It starts with the initial set of available discriminatory paths and generates other inputs belonging to the nearby execution paths, thereby systematically performing local explanation while banking on the adversarial robustness property. Our experimental evaluations clearly show that our approach performs better than all the existing tools.

Symbolic testing of machine learning models is bound to pave way for a number of efforts in future, some of them are listed below.
A generic framework for testing. Our global search method is a generic way of testing any black-box ML model and not just a method towards solving individual discrimination. For a different problem, there may be a need to change the implementation of only `check_for_error_condition` function, especially if the modality remains the same.
Local Model Generation. In this paper, we are very much dependent on LIME which is a local model generator. The approximation caused by the local model does have a role to play as far as the effectiveness of systematic exploration is concerned. In future, we want to investigate how we can generate a more effective and efficient local model for better exploration.
Model Path Definition. For testing neural networks, we can use neuron activation to precisely define *path.*
Global Approximation. We have used a local approximation as a way to get an interpretable portion of the model. Intuitively, that decision goes with the 'on-demand exploration' strategy which does not require re-engineering of all the parts of a model. Also, local model generation does not require any training data. In future, we also wish to investigate the global approximation algorithm like TREPAN [3] in order to create decision tree models and perform symbolic execution on such a decision tree in case of availability of training data.
Adversarial Robustness. There exists many white-box approaches addressing the problem of finding adversaries. In future, we may investigate if such techniques can be applied in case of black-box testing to detect individual discrimination.
Hybrid Search Strategy. Our algorithm gives preference to local search over the global one, and seed data over global search. Finding an interleaved strategy can be another direction to pursue in future.
Explaining Source of Individual Discrimination and De-Biasing. Once an individual discrimination is detected in a model, we would like to uncover its root cause, especially by mapping it to the training data. We envision the use of influence function [11] to achieve this. Once we figure out the training data instance and their attributes responsible for bias, it is then possible to de-bias the model by either removing or appropriately perturbing the training data instances.

REFERENCES

[1] Julius Adebayo and Lalana Kagal. 2016. Iterative Orthogonal Feature Projection for Diagnosing Bias in Black-Box Models. arXiv:arXiv:1611.04967

[2] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

[3] Mark William Craven. 1996. *Extracting Comprehensible Models from Trained Neural Networks*. Ph.D. Dissertation. AAI9700774.

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[5] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness Through Awareness. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. ACM, New York, NY, USA, 214–226. https://doi.org/10.1145/2090236.2090255

[6] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 259–268.

[7] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness Testing: Testing Software for Discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 498–510. https://doi.org/10.1145/3106237.3106277

[8] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 47–54. https://doi.org/10.1145/1190216.1190226

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[10] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S. Pasareanu, and Sarfraz Khurshid. 2018. Symbolic Execution for Deep Neural Networks. arXiv:arXiv:1807.10439

[11] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1885–1894. http://proceedings.mlr.press/v70/koh17a.html

[12] Matt Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual Fairness. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., USA, 4069–4079. http://dl.acm.org/citation.cfm?id=3294996.3295162

[13] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 120–131. https://doi.org/10.1145/3238147.3238202

[14] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[15] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. 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*. ACM, 1135–1144.

[16] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. 2017. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 618–626. https://doi.org/10.1109/ICCV.2017.74

[17] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[18] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. arXiv:arXiv:1803.04792

[19] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. arXiv:arXiv:1805.00089

[20] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *CoRR* abs/1312.6199 (2013). arXiv:1312.6199 http://arxiv.org/abs/1312.6199

[21] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[22] F. TramÃĺr, V. Atlidakis, R. Geambasu, D. Hsu, J. Hubaux, M. Humbert, A. Juels, and H. Lin. 2017. FairTest: Discovering Unwarranted Associations in Data-Driven Applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 401–416. https://doi.org/10.1109/EuroSP.2017.29

[23] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated Directed Fairness Testing. Automated Directed Fairness Testing. In Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 18), September 3-7, 2018, Montpellier, France. (2018). https://doi.org/10.1145/3238147.3238165 arXiv:arXiv:1807.00468

[24] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 408–426.