Aequitas: Automated Fairness Testing on Machine Learning Database

CS Comps Winter 2022

By: Yemi Shin, Yunping Wang, Michael Worrell, and Juanito Zhang Yang

Advisor: Dave Musicant

Table of Contents

Theory and Applications of Machine Learning Fairness: Review from the Perspective of Fairness Tester *Aequitas*

Yemi Shin, Yunping Wang, Michael Worrell, and Juanito Zhang Yang
Department of Computer Science, Carleton College, 300 North College Street, Northfield, MN, 55057

(Dated: March 16, 2022)

In recent years, an increase in application for machine learning gave rise to the discipline of machine learning fairness. Machine learning fairness is the study that aims to reduce the inherent discrimination present in all models. This survey paper aims to explore the theory and application of machine learning fairness through the lens of the algorithm *Aequitas*. *Aquitas* is a preprocessing bias-correcting algorithm that implements individual and counterfactual fairness. To understand the algorithm We first address the theory of machine learning and give a overview of the landscape of fairness. After that, the algorithm *Aquitas* is studied in more detail, especially its mathematical background and code implementation.

CONTENTS

I. INTRODUCTION

In recent years, machine learning has gradually become an integral tool in our everyday lives. Machine learning is used to determine whether a bank should issue a loan to someone[1], a company should hire someone[2], and in certain extreme cases, how long someone's criminal sentence should be[3,4]. However, this up-ticking in range of application for machine learning is not necessarily a net positive for the society. In particular, since machine learning refers to the process of using real-world data to improve upon a computer program, pre-existing social biases in those data may carry through the resulting model. These biases may then be propagated further when biased models are used in real world applications, creating potential discrimination[5].

Thus, a new field of study is emerging that aims to counteract these biases, namely machine learning fairness, a discipline where we examine the intrinsic biases in machine learning models and devise algorithms to adjust our model to be resilient to these biases. One of these algorithms is call *Aequitas*, developed by Udeshi et al and presented in the paper *Automated Directed Fairness Testing*[6].

In this paper, we will give an overview of the field of machine learning fairness and explore *Aequitas* in more detail. In the first half of the paper, we will provide the necessary background information to understand the algorithm and its significance in the overall machine learning fairness landscape. In the second half, we will provide an overview and explanation of the *Aequitas* algorithm.

We will start our discussion by developing the theoretical framework of machine learning that can aid us in understanding *Aequitas*. In Section II, we will discuss the basic theory of machine learning. We will then develop the theory of training machine learning models in Section III. Building on that, in Section IV, we will discuss three machine learning models that are relevant to *Aequitas*: decision tree, random forest and support vector machines (SVM). This will conclude our excursion into the theory of machine learning.

Starting from Section V, we shift our focus to machine learning fairness. In this section, we aim to give an overview of the field of machine learning fairness by discussing different definitions of fairness and how they can be applied to various circumstances. We will also discuss common strategies that are used to enforce these types of fairness in a machine learning model. This section will provide background information for us to place *Aequitas* within the larger field of fairness and pinpoint the exact problem that it tries to address.

The last necessary piece of information that is needed will be the mathematical principle behind *Aequitas*'s operation. *Aequitas* leverages two main mathematical concepts in its implementation: robustness in machine learning model and the law of large numbers. We will discuss the rigorous details of machine learning robustness in Section VI and law of large numbers in Section VII.

After presenting all the relevant information needed to understand *Aequitas*, we will explore the algorithm itself in Section IX. The original author also provides a proof of concept implementation for the algorithm in Python, which we will discuss in the Appendix.

II. FUNDAMENTALS OF MACHINE LEARNING

To start the discussion of the theory behind machine learning, we will need to define what learning is. Learning occurs when an agent improves its performance after making an observation about the world. An agent can make observations through a data set. A data set is a set of input-output pairs. In this paper, the input in the data set will be in factored representation. A factored representation refers to a vector of attribute values $(x_1, x_2, \ldots, x_n)$. As for the output, we will focus on two main types of output: classification and regression. A classification is when an output is one of a finite set of values, e.g., true/false. A regression is when the output is a continuous number[7].

We characterize the idea of learning through the concept of feedback. Specifically, we divide machine learning algorithms into three sub-categories: 1), Supervised Learning, 2), Unsupervised Learning, and 3), Reinforcement Learning. We call a learning process supervised if the agent observes input-output pairs and learns a function that best maps input to output. We call a learning process unsupervised if the agent learns patterns without explicit feedback from the programmer. Finally, we call a learning process reinforced if the agent learns from a series of reinforcement taking the form of rewards and punishment. In this paper, we will focus our attention on supervised machine learning.

A. Supervised Learning

Here we provide a formal definition of supervised learning:

Definition II.1 (Supervised Learning)**.** [7] Given a training set of N example input-output pairs

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n),$$

where each pair is generated by an unknown function $y = f(x)$, discover a function h that approximates the true function y.

We call h the hypothesis function about the world (or model of the world), and we define the function space of all possible h as the hypothesis space/model class $\mathscr{H}$. We also define the output y_i as the ground truth.

With Definition II.1, we are equipped to define a consistent hypothesis:

Definition II.2 (Consistent Hypothesis)**.** A hypothesis h is consistent on a training set N if $\forall (x_i, y_i) \in N$, $h(x_i) = y_i$.

To evaluate a hypothesis, we will test it on a test set T that is different from the training set N. A hypothesis generalizes well if $h(x_j) = y_j \forall (x_j, y_j) \in T$.

To analysis how well a hypothesis h generalizes, we will introduce two ideas: bias and variance, which in turn defines the concept of underfitting and overfitting.

Bias refers to the tendency of a predictive hypothesis to deviate from the expected value when averaged over different training set. In other words, a hypothesis that is high on bias has tendencies to ignore patterns in data sets. We say that the hypothesis is underfitting the data. Bias often results from strong restrictions placed over the hypothesis space $\mathscr{H}$. For example, a hypothesis space of linear function induces a strong bias as any hypothesis generated will only captures the overall slope of the data over time and lose any other patterns.

Variance refers to the tendency for the hypothesis to exhibit massive change from small fluctuations in the training set. Alternatively, a hypothesis with high variance pays too much attention to details that are specific to a particular training set and fails to generalize over unseen data. We characterize such hypothesis as overfitting. A typical example of a hypothesis space that tends to overfit is a high-degree polynomial.[7].

III. DEVELOPING MACHINE LEARNING MODELS

The goal of machine learning is to select a hypothesis that will optimally fit unseen future examples. However, in order to parse this statement, we would first need to specify the meaning of optimal fit and the restrictions on future examples[7].

We first need to place a restriction onto the dataset. Here we are assuming that all data a randomly drawn from a population with a probability distribution P. We claim that all future examples must satisfies the stationary assumption. The definition of stationary assumption is stated below:

Definition III.1 (Stationary Assumption)**.** We assume that for an arbitrary data set E, the following are true $\forall E_j \in E$:

i), All E_j has the same prior probability distribution, or

$$P(E_j) = P(E_{j+1}) = P(E_{j+2}) = \ldots;$$

ii), All examples are independent of the previous examples, or

$$P(E_j) = P(E_j|E_{j-1}, E_{j-2}, \cdots).$$

In other words, all examples need to be independent and identically distributed[7].

To define an optimal fit, we first introduce the concept of error rate, which is the proportion of time that $h(x) \neq y$ for an (x,y) example. We define an optimal fit as the hypothesis that minimizes error rate. Since a model is based on a training set, we need to evaluate the model on an alternative data set to make sure that our model is not overfitting. Thus, we will need a test set comprised of different examples from the training set.

Before we proceed, we also need to define the idea of a hyperparameter. A hyperparameter is a parameter that determines how the model is generated. In other words, it is a parameter for the model training itself. Therefore, when generating a model, the programmer has control over the hyperparameters.

To summarize, we can write down the following simple process for supervised learning:

1. Use a training set to train the data
2. Adjust the hyperparameters for the model
3. Test the data on a validation set to see whether the adjustment improved the model or not
4. Repeat steps 1 - 3
5. Evaluate the final model on a test set

Note that the validation set here refers to a data set that is different from both the training set and the test set. The reason for needing a validation set is that we need to evaluate each model independently of the data set.

We can break down the task of finding a hypothesis into two sub-tasks: 1), Model selection, which refers to the process of choosing a hypotheses space, and 2), Optimization/Training, which finds the best hypothesis in the space. The study of model selection is beyond the scope of this paper is best addressed in textbooks such as Stuart Russell and Peter Norvig's *Artificial Intelligence: A Modern Approach*[7].

IV. COMMON MACHINE LEARNING MODELS

In this section we will be discussing some common machine learning models and their training processes.

A. Decision Tree

A decision tree is a Boolean classifier that maps a vector of attributes to true/false. In other words, a decision tree hypothesis takes the form

$$h : A^n \rightarrow \{true, false\}, \tag{1}$$

where elements of A^n are lists with n elements.

We can modify a Boolean classification into a decision tree using the following structure: a node in the decision tree is a test of a single input attribute. A branch is labeled with possible values of the attribute, and a leaf is a specific classification that the tree will return. A decision tree thus reaches its decision by passing through a series of tests, starting from the root until it reaches one of its leaves. In the remaining sections, we will consider any example with $true$ output as a positive example, and $false$ output as a negative example.

Note that we can rewrite a Boolean decision tree to the following equivalent logical statement:

$$Output \Leftrightarrow (Path_1 \vee Path_2 \vee Path_3 \vee \ldots), \tag{2}$$

where each $Path_i$ is a conjunction of the form $(A_m = v_x \wedge A_n = v_y \wedge \ldots)$ of attribute-value tests that correspond to a path from the root to a $true$ leaf. The whole expression is then in disjunctive normal form. Thus, any function of propositional logic can be turned into a decision tree.

To generate a decision tree from a training set, we will use a greedy divide-and-conquer algorithm. The main idea behind the algorithm is that we want to test the attribute with the largest information gain first, and then recursively solve the smaller sub problems. The full algorithm is given below:

1. If the examples are all positive or negative, we are done.
2. If there are both positive and negative examples, then choose the attribute with the highest *IMPORTANCE* to split them. Then recursively continue testing the subsets.
3. If the are no examples, it means that no examples are in the train set with this combination of attributes values, and we return the most common output value from the training set that were used in constructing the node's parent.
4. If there are no attributes, but still both positive and negative examples, it means that these examples have the same exact description, but different Boolean classifications. In other words, the model gave two identical example different classifications. This can happen due to error or noise in the data. The best we can do is return the most common output value from the remaining sets.

Now we need to define the *IMPORTANCE* function. To do that, we need to introduce the concept of information entropy. Entropy measures the uncertainty of a random variable. Thus, more information correlates to less entropy. For example, a random variable with one possible outcome has zero entropy. Formally, we define entropy of a random variable V as

$$H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k). \quad (3)$$

For a Boolean random variable that is true with probability q, we define its entropy $B(q)$ as

$$B(q) \equiv -(q \log_2 q + (1-q) \log_2 (1-q)). \quad (4)$$

Therefore, for a training set with p positive examples and n negative examples, the total entropy would be

$$H(Output) = B(\frac{p}{p+n}). \quad (5)$$

Moreover, an attribute A with d distinct values divides the training set E into subsets $E_1, \ldots, E_d$. Each subset E_k has p_k positive examples and n_k negative examples. Formally, we define E_k as

$$E_k = \{e \in E : A(e) = A_k\}, \quad (6)$$

where $A(e)$ is the value of attribute A for example e. Thus, if we know $A = A_k$, we still need additional

$$H(Output|A = A_k) = B(\frac{p_k}{p_k + n_k}) \quad (7)$$

bits of information to determine whether a given example is positive or negative.

A randomly chosen example from the training set would have the kth value for an attribute. In other words, it will be in set E_k with probability

$$P(e \in E_k) = \frac{p_k + n_k}{p+n} = P(A = A_k). \quad (8)$$

Combining these two results, we can determine the entropy of the tree after we test for attribute A, or

$$Remainder(A) = H(Output|A). \quad (9)$$

We know that

$$H(Y|X) = \sum_x P(X = x) H(Y|X = x). \quad (10)$$

Using our previous results, we then have

$$H(Output|A) = \sum_{k=1}^{d} P(A = A_k) H(Output|A = A_k) \quad (11)$$

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B(\frac{p_k}{p_k + n_k}) \quad (12)$$

Now we can define the information gain from the attribute test on A as the expected reduction in total entropy, or

$$Gain(A) = H(Output) - Remainder(A) \quad (13)$$

$$\Rightarrow \quad Gain(A) = B(\frac{p}{p+n}) - Remainder(A) \quad (14)$$

Finally, we can formally define a way to quantify the *IMPORTANCE* of attributes A as $Gain(A)$.

However, a decision tree algorithm only finds a hypothesis that best fits the training data, when what we really want is to generalize for unseen data. With higher attribute count, we are much more likely to overfit. Therefore, we would need to introduce the concept of decision tree pruning. Pruning works by eliminating nodes that are clearly not relevant. We start with the full tree as last time, but this time we look at test nodes with only leaf nodes as their descendants. If a test is irrelevant, we replace the test node with a leaf node. Pruning continues until all test nodes with only leaf nodes as their descendants are either pruned or accepted as they are.

We now need to decide how to determine whether a test node is relevant. To do this, we use a statistical significance test. We start by assuming that there are no underlying patterns (Null hypothesis). We then compute to what extent the actual data deviates from a total lack of pattern. If the degree of deviation is statistically unlikely($> 5\%$), it signifies that there is a significant pattern in the data.

In our case, if we have $v = p+n$ examples, we can compute the expected positive $\hat{p_k}$ and negative $\hat{n_k}$ examples in each subset E_k using the overall ratios of p and n, and compare them to the actual p_k and n_k. Thus, we define $\hat{p_k}$ and $\hat{n_k}$ as such

$$\hat{p_k} = p \times \frac{v_k}{v} = p \times \frac{p_k + n_k}{p+n}, \quad (15)$$

$$\hat{n_k} = n \times \frac{v_k}{v} = n \times \frac{p_k + n_k}{p+n}. \quad (16)$$

We can then compute the deviation using a χ^2 test:

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p_k})^2}{\hat{p_k}^2} + \frac{(n_k - \hat{n_k})^2}{\hat{n_k}^2} \tag{17}$$

This equation follows the χ^2 distribution with $d-1$ degrees of freedom. Using this distribution, we see that $\Delta = 7.82$ would reject null hypothesis at 5% level, and value below are accepted. This technique is called χ^2 pruning. With this, our construction of a decision tree is complete.

B. Support Vector Machines

A support vector machines(SVM) is a non-parametric model that attempts to generate a maximum margin separator. A maximum margin separator is a decision boundary where every example on one side of the boundary will have the classification 1, and every example on the other side will have the classification -1. We need to develop an algorithm that generates such decision boundary. Note that since we do not constrain the dimension of the attribute space, our decision boundary will be a hyperplane defined by

$$\{\vec{x} : \vec{w} \cdot \vec{x} = b\}, \tag{18}$$

where $\vec{w}$ and b are the coefficients that we need to find. We will also define the distance between the decision boundary and the nearest example as d. See Fig. 1 for a two-dimensional illustration of the problem.

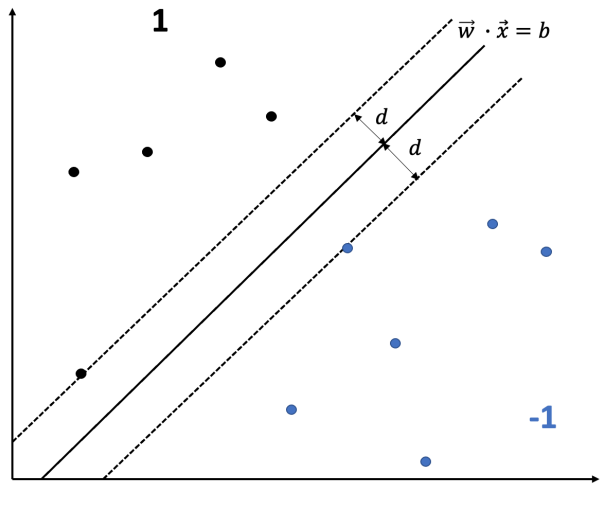

Figure 1. A two-dimensional SVM problem. The top half are the examples with classification 1, and the bottom half are the example with classification -1. The line in the center is the decision boundary with a separation of d to the nearest examples.

Observe that

$$\forall \vec{x} \in \mathbf{1},\ \vec{w} \cdot \vec{x} > b + d, \tag{19}$$

and

$$\forall \vec{x} \in -\mathbf{1},\ \vec{w} \cdot \vec{x} < b - d, \tag{20}$$

where $\mathbf{1}$ is the set of examples with classification 1, and $-\mathbf{1}$ is the set of examples with classification -1.

We can now define y_i such that for every x_i:

$$y_i = \begin{cases} 1, & x_i \in \mathbf{1}, \\ -1, & x_i \in -\mathbf{1}. \end{cases} \tag{21}$$

Then, we can combine Eq. 19, 20 and Eq. 21 to get

$$y_i(\vec{w} \cdot \vec{x} - b) < d. \tag{22}$$

We can also show that

$$d = \frac{2}{||\vec{w}||}. \tag{23}$$

Then, we can rearrange this inequality to get

$$y_i(\vec{w} \cdot \vec{x_i} - b) < 1 + e_i, \tag{24}$$

where e_i is the error of the decision boundary, or the number of examples that are placed on the wrong side. Therefore, this problem has been reduced to a problem of finding

$$\min(\sum_{i=0} e_i + ||\vec{w}||^2). \tag{25}$$

This is a quadratic programming optimization problem, and the detail of such algorithm is beyond the scope of this paper. After solving this for $\vec{w}$, we will have our decision boundary.

C. Random Forest

To discuss random forest classifier, we first need to introduce the concept of ensemble learning. Ensemble learning is the process of selecting a collection of hypotheses(base model) and combining them to create a joint-decision model. The main reason for using ensemble learning is that it reduces bias and variance in our hypothesis.

Random forest is a form of ensemble learning that is based on a technique called bagging. In bagging, K distinct training sets are generated by randomly sampling N examples with replacement from the original training set. A machine learning algorithm is then run on the sub-training set to get K hypotheses. These hypotheses are aggregated to make the final prediction. For classification problems, we use plurality vote; for regression problems, we compute the average:

$$h(x) = \frac{1}{K} \sum_{i=1}^{K} h_i(x). \tag{26}$$

Bagging tends to reduce variance and is the standard approach when the base model overfits.

Random forest is a form of decision tree bagging in which we take extra steps to make the ensemble of K trees more diverse. The key idea is to randomly vary the attribute choice rather than the training examples. At each point in the construction of the tree, we select a random sampling of n attributes, and compute the one that yields the most information gain. Consequently, random forests are not pruned and are resistant to overfitting. As more trees are added to the forest, the error converges.

V. FAIRNESS IN MACHINE LEARNING

It can be challenging to know what exactly machine learning fairness is. The field of fairness in machine learning has been growing in size, coverage, and importance. It has been growing into its own conferences, and many new papers on machine learning fairness are released each year.

When relying on the use of software to calculate outcomes for complicated decision making tasks, it is easy for unintentional bias and discrimination to make their way into more and more elements of everyday life. This could make it so that decision making processes thought of to be fair and unbiased could produce inaccurate and unfair decisions.

The consequences of machine learning making mistakes when it comes to fairness are important to consider when dealing with machine learning. One notable recent example is the COMPAS computer program, which was used to assist in sentencing defendants, which was eventually determined to be making unfair decisions. After filtering criminal history, COMPAS was 77% percent more likely to incorrectly flag black defendants as high risk than it was likely to falsely flag white defendants, and was also more likely to falsely identify white individuals as low-risk.[8]

These kinds of problems require technical and legal definitions of fairness, as well as definitive ways to improve the fairness of machine learning models. However, defining, measuring, and improving fairness is a long-standing problem that has been debated and discussed for decades, long before machine learning reached the prevalence it has today. One indicator of the current state of machine learning fairness is how fairness is determined legally. Some of the modern ways that laws evaluate the fairness of decision making processes are via the presence of *disparate treatment* and *disparate impact* present in or resulting from a particular decision making process.[9]

Disparate treatment is the direct use of sensitive attribute data in the decision making process, while disparate impact is the presence of negative impacts resulting from the decision making process. Attempting to circumvent either disparate treatment or disparate impact on their own, by methods such as not considering the sensitive attribute to avoid disparate treatment, or by using sensitive attributes to avoid disparate impact, can lead to other forms of discrimination (Indiscrete Discrimination and Reverse Discrimination being a few notable examples).[9]

A. Defining Fairness

One thing we took away from our research is that one cannot guarantee the ability to maximize multiple conflicting definitions of fairness simultaneously. Because of this, those that measure machine learning fairness need to make a choice of how one will define the fairness of a given model prior to analyzing its fairness.[10]

Definitions of fairness attempt to help mediate a problem specific to their definition. There have been multiple definitions of fairness proposed throughout the past several decades. These definitions work to help individuals maximize different categories of fairness, made to provide fairness in a specific context . A few of these categories include individual fairness, non-comparative fairness, subgroup parity, and correlation.[10]

With Aequitas, we are catering to the category of individual fairness. Aequitas attempts to discover discriminatory input (inputs that violate individual fairness). Individual fairness attempts to make sure that, for two entries that are identical with the exception of some sensitive attribute (or identical within a specific input space), the fairness model will output identical results for both entries[6]. For Aequitas, the precise definition of (individual) fairness is as follows:[6]

Definition V.1. Let f be a classifier under test, γ be the predetermined discrimination threshold (e.g. chosen by the user), $\mathbb{I}$ be the input domain of the model, P_i be the i-th input parameter of the model, P be the set of all input parameters, P_{disc} be the non-empty set of all potentially discriminatory parameters, $I \in \mathbb{I}$, and I_p be the value of input parameter p in input I. Assume $I' \in \mathbb{I}$ such that there exists a non-empty set $Q \subseteq P_{disc}$ and for all $q \in Q$, $I_q \neq I'_q$ and for all $p \in P \backslash Q$, $I_p = I'_p$. If $|f(I) - f(I')| > \gamma$, then I is called a discriminatory input of the classifier f and is an instance that manifests the violation of (individual) fairness in f.

This provides Aequitas with a method to check whether a given model produces what it considers unfair classifications.

B. Measuring Accuracy

There are multiple metrics that can be used to measure the performance of a machine learning fairness model. This includes fairness (exact definition subject to the model), but another way that machine learning fairness can be measured is by looking at the accuracy of the model based on classifier label assignments. The accuracy of a given model can be measured in terms of correct assignments, false positives, and / or false negatives relative to the total number of results outputted by the model. When working with models whose classifiers assign a binary decision -1 or $+1$ (for example, whether or not an individual is likely to succeed at Carleton College, in order to determine whether a student should be accepted), an incorrect assignment can either be a false positive or a false negative (in which an entry is given the binary outcome $+1$ when it should have returned -1, and vice versa). These rates can differ for different attributes.

The distributions of false positives and false negatives within a given dataset can sometimes provide more information about the fairness of a model than considering both as a single measurement. While the accuracy of COMPAS was approximately the same for different racial groups, the false positive and false negative rates differed widely between the racial groups.[8]

From a theory perspective, unless you have access to perfect information, you cannot make sure that all three forms of accuracy are equally treated. Because it is so challenging for all three factors to be treated equally, different groups and in-

dividuals have proposed different ideas over which elements of scoring the accuracy of fairness should be prioritized.

Aequitas may fail to locate all discriminatory inputs in the input space; however, it does guarantee that no false positives (non-discriminatory inputs that are incorrectly marked as being discriminatory) will be generated. This needs to be taken into account when considering the accuracy of the model.

C. Fairness / Accuracy Trade-offs

One method to deal with the trade-off between fairness and accuracy is via the use of fairness constraints as well as via the use of accuracy constraints. These constraints allow a minimal degree of leeway in either accuracy or fairness, as long as the other remains at a predetermined level deemed acceptable. One important accuracy constraint is the business necessary clause, and one important fairness constraint is the p% rule.

The business necessity clause maximizes fairness under accuracy constraints. Because corporations, academic institutions, and other groups are often required to meet performance standards when it comes to the accuracy of their decision making processes, they are allowed to exhibit some degree of unfairness in order to meet these accuracy thresholds.[9]

While Aequitas aims to increase accuracy when possible, it instead works more to maximize accuracy under fairness constraints. It does this using a commonly-used fairness constraint present in multiple previous models: the p% rule.[6]

A given decision making process fails to satisfy the p% rule if the proportion of individuals with some attribute that receive some classification (positive or negative) is less than p% of the proportion of individuals without the attribute that receive the other classification. Decision making processes in violation of this p% rule are likely to be making biased decisions. However, if a committee decides to investigate a company in violation of this rule, said company only need to provide a reasonable explanation for this violation as a justification if it occurs. This rule has been occasionally altered to fit different kinds of models, such as working alongside a decision boundary.[9]

While Aequitas states that it can deal with discriminatory input using any p% rule, the authors of the Aequitas paper only provided the results when using a 100% rule, by setting their discriminatory threshold used in testing to 0.[6] While this greatly increases the fairness present in the retrained dataset, having such a high p% rule value could potentially lead to high levels of inaccuracy.[9]

D. Machine Learning Fairness Strategies Overview

Different machine learning strategies use different methods to deal with issues of fairness. On a high level, these fall into the following categories: preprocessing, processing, and post-processing models.

Aequitas is a pre-processing model that discovers discriminatory inputs and can retrain a model based off of the discriminatory inputs discovered. Therefore, to better understand Aequitas, it is relevant to first dive deeper into the purposes and mechanics behind pre-processing, as well as how Aequitas retraining differs from previous approaches at machine learning fairness.

E. Preprocessing

Preprocessing assumes that unfairness stems from biased data that fails to reflect reality. Some argue that preprocessing is correcting historical biases. In other words, preprocessing attempts to deal with the truth that not all data reflects the reality of the world, because assumptions have been made that led to an under-representative dataset. Preprocessing is a common method amongst machine learning fairness models, yet different models rely on different assumptions about how the data being provided reflects the real world.

One model around causality is counterfactual fairness. Counterfactual fairness assumes that if some discriminatory attribute should not be causally linked to a given outcome in reality, then changing said attribute will not change the outcome in a fair model. If the data demonstrates an effect between the input and output, then the model is deemed to be unfair. If the assumption being made reflects reality, then counterfactual fairness is likely to identify discriminatory inputs. At a high level, Aequitas runs on the assumption that the discriminatory input being discovered (in their case, gender) should not have a causal effect on the results if all other attributes are shared between two entries.[6]

F. Retraining

One of the things that sets Aequitas apart from other machine learning fairness models is the fact that Aequitas is capable of using found discriminatory inputs in order to retrain a model, making it possible for the retrained model to exhibit a higher degree of fairness.[6] Aequitas uses a directed test generation strategy to assist in retraining classifiers. More information on Aequitas retraining can be found in the "Understanding Aequitas" section of this survey paper.

Many of the previous strategies in machine learning fairness initially find discriminatory input based on random points in the input space. Aequitas also finds discriminatory input based on random points in the input space. However, unlike previous methods, when Aequitas identifies said input, it attempts to find related points in the vicinity of the said inputs that might also be discriminatory. This deals with a common problem other programs face: random test generation risks not identifying discriminatory input when all the discriminatory input is clumped in a few regions.[6] Aequitas then uses the results of the directed test generation to retrain existing models with an increasing percentage of discriminatory input added to the training set, as long as the fairness of the model increases and the percentage does not exceed 100%. One thing to consider is that the research presented by the primary source assumes the robustness property of machine learning models is

true. More information on robustness property can be found in Section VII of this survey paper.

G. Limitations of Machine Learning Fairness Models

Previous machine learning fairness strategies were not perfect, and were fraught with limitations. Some of them also apply to Aequitas, while for others Aequitas managed to find new strategies to work around them. One limitation faced by a wide breadth of machine learning fairness models is that each only worked with a narrow range of classifiers, and not all previous models were able to eliminate both disparate impact and disparate treatment simultaneously. Another limitation of previous machine learning fairness strategies is that there are often limits in the types of sensitive attributes the methods can handle. These can include only working with a single attribute, or not being able to handle non-binary sensitive attributes.[9] The primary source believes that Aequitas could be expanded to work on different types of sensitive attributes. At the moment, Aequitas only works on one discriminatory input.

H. Further Resources

This fairness in machine learning section of the survey paper has provided a brief overview of some of the topics of fairness preceding and relevant to Aequitas, but there is more information available for further learning that did not get covered in this limited amount of space. Here are a few:

- For more information on more types of fairness that have been proposed over the past several decades, as well as more information about fairness and unfairness both in and out of the field of machine learning, see B. Hutchinson and M. Mitchell (2019).
- More information on the strengths and limitations of previous models, as well as more information on fairness constraints, can be found from Zafar M, Valera I, Rodriguez M et al. (2015).

VI. ROBUSTNESS IN MACHINE LEARNING

As we will see when we explain *Aequitas* in detail, the principal underlying assumption of the *Aequitas* algorithm is the robustness of machine learning models. Without this hypothesis, we cannot generate similar discriminatory inputs based on the initial set of discriminatory inputs found on the first part of the algorithm (see Section IX). Even though we will give the precise and technical definition of robustness, we can understand, for now, that robustness states that the output of a machine learning model is not dramatically changed by small changes to its input.

The robustness claim made by our primary source is based on the results of Fawzi et. al (2015)[11], which we describe in this section.

Adversarial perturbations are minimal perturbations made to the inputs of a classifier so that the classifier will switch the estimated label of that input. The opposite of adversarial perturbation is random uniform noise, which refers to randomly changing an input, in contrast to making changes to intentionally change the label of the input. The main result from this paper is that there is an upper bound to classifiers' robustness against adversarial perturbations and random uniform noise.

The paper also claims that the adversarial instability is due to the low flexibility of some classifiers. This refers to the capability of expression of some model classes, for instance, linear classifiers are less flexible than high-degree polynomial classifiers.

A. Definitions

Now, we begin introducing the technical terms that will allow us to formally express the aforementioned upper bound.

Let μ be a probability measure defined on $\mathbb{R}^d$, which assigns probabilities on the subset of points that we wish to classify. For each point x in this subset, let y be the unknown function that we want to approximate and such that $y(x) \in \{-1, 1\}$. Then μ tells you the probability that $y(x) = 1$ or $y(x) = -1$ for any x that we wish to classify. Furthermore we assume that these points x are in a M-ball,

$$\mathbf{B} = \{x \in \mathbb{R}^d : \|x\|_2 < M\}.$$

Note that,

$$P_\mu(x \in \mathbf{B}) = 1.$$

Now let μ_1 and μ_{-1} be the probability distribution of points $x \in \mathbf{B}$ such that $y(x) = 1$ and $y(x) = -1$, respectively. Furthermore, suppose that $f : \mathbb{R}^d \to \mathbb{R}$ is a classification function where the sign of $f(x)$ is the label that we will assign to $x \in \mathbb{R}^d$. Note that in the language that we had previously, f is the hypothesis, while y is the unknown "truth of the world."

One common measure of performance for f is the risk of f,

$$\begin{aligned} R(f) &= P_\mu(\text{sign}(f(x)) \neq y(x)) \\ &= p_1 P_{\mu_1}(f(x) < 0) + p_{-1} P_{\mu_{-1}}(f(x) \geq 0), \end{aligned} \tag{27}$$

where $p_{\pm 1} = P_\mu(y(x) = \pm 1)$. The non-technical way to express the risk of f is "what is the probability that we sample from $\mathbf{B}$ by μ and get a point x that f will missclassify."

Given $x \in \mathbb{R}^d$ sampled with probability as determined by the probability measure μ, we define $\Delta_{\text{adv}}(f : x)$ as the smallest perturbation that changes the sign of $f(x)$. In other words,

$$\Delta_{\text{adv}}(f : x) = \min_{r \in \mathbb{R}^d} \|r\|_2 \text{ with } f(x)f(x+r) \leq 0. \tag{28}$$

The robustness to adversarial perturbations of f is defined as the average $\Delta_{\text{adv}}(f : x)$ over all $x \in \mathbf{B}$,

$$\rho_{\text{adv}}(f) = E_\mu(\Delta_{\text{adv}}(f : x)). \tag{29}$$

Furthermore, note that this value is independent of the actual signs of $f(x)$ or $y(x)$, rather it depends solely on how much we need to perturb the inputs to change the estimated label.

We can similarly define robustness of f to random uniform noise. Given $\varepsilon \in [0,1]$, let,

$$\Delta_{\text{unif},\varepsilon}(x:f) = \max_{\eta \geq 0} \eta \text{ with } P_{n \sim \eta \mathbb{S}}(f(x)f(x+r) \leq 0) \leq \varepsilon \tag{30}$$

where $\eta\mathbb{S}$ is the η-Ball around x. Intuitively, this value measures the maximum radius around x for which the probability of missclassifying a point within the ball defined is less than ε. Similarly, the robustness is the average $\Delta_{\text{unif},\varepsilon}(x:f)$ over all $x \in \mathbf{B}$,

$$\rho_{\text{unif},\varepsilon}(f) = E_{\mu}(\Delta_{\text{unif},\varepsilon}(f:x)). \tag{31}$$

It is important to note that there are classifiers f such that $R(f) = 0$ for which $\rho_{\text{adv}}(f)$ is small. In other words, the fact that a classifier has, in particular, 0 risk on its training set doesn't mean that it will be robust against adversarial perturbations. This is akin to the problem of over-fitting, where a classifier works very well for its training dataset, but it doesn't capture the concept of the problem correctly.

B. Assumption A

We say that a classifier f satisfies Assumption A if there are real numbers $\tau > 0$ and $0 < \gamma \leq 1$ such that for every $x \in \mathbf{B}$,

$$\text{dist}(x, S_-) \leq \tau \max(0, f(x))^{\gamma} \tag{32}$$

$$\text{dist}(x, S_+) \leq \tau \max(0, -f(x))^{\gamma} \tag{33}$$

where

$$\text{dist}(x, S) = \min_{y} \{\|x - y\|_2 : y \in S\}$$

and $S_+ = \{x \in \mathbf{B} : f(x) \geq 0\}$ and $S_- = \{x \in \mathbf{B} : f(x) < 0\}$.

Intuitively, we say that f satisfies Assumption A if for any $x \in \mathbf{B}$ the distance from x to S_- and S_+ is bounded.

C. Upper Bound

If f satisfies Assumption A then,

$$\begin{aligned}\rho_{\text{adv}}(f) \leq 4^{1-\gamma}\tau(&p_1 E_{\mu_1}(f(x)) \\ &- p_{-1}E_{\mu_{-1}}(f(x)) + 2\|f\|_{\infty}R(f))^{\gamma}.\end{aligned} \tag{34}$$

Therefore, we have found an upper bound of machine learning model's robustness against adversarial perturbations. This bound is done with respect to the risk as well as the difference between the expectations of the values of the classifiers computed on the distributions μ_1 and μ_{-1}. We will not worry about what the value of the right hand side of the inequality is, just that the value $\rho_{\text{adv}}(f)$ is bounded.

This ends our survey on this topic, however there is more about robustness of linear and quadratic classifiers in Fawzi et al (2015). There is also a proof of equation (34) and a similar upper bound for robustness of linear classifiers against random uniform noise. Furthermore the paper goes on to explain that, for the particular case of linear classifiers, the robustness against random uniform noise is greater than the robustness against adversarial perturbations by a factor of $\sqrt{d}$, where d is the dimension of the inputs.

VII. LAW OF LARGE NUMBERS

Aequitas also relies on the Law of Large Numbers, as the idea behind the Global search step is that repeated sampling of the input space guarantees that at least one discriminatory input will be found.

Thus, as a group, we decided to investigate more about this concept, and this is what we found[12].

A. Weak Law of Large Numbers

Let $(X_1, X_2, \ldots)$ be an independent and identically distributed sequence of random variables with finite expectation ν. That is $E(X_1) = E(X_2) = \cdots = \nu$. The sequence of sample averages $(\overline{X}_1, \overline{X}_2, \ldots)$, where,

$$\overline{X}_i = \frac{X_1 + \cdots + X_i}{i}$$

satisfies,

$$\lim_{n \to \infty} P(|\overline{X}_n - \nu| < \varepsilon) = 1 \ \ \forall \varepsilon > 0. \tag{35}$$

What this means is that there is a very high probability that the sample mean is within an arbitrary margin of the expected mean as the number of experiments increases.

B. Strong Law of Large Numbers

With the same notation as the Weak Law of Large Numbers, we claim,

$$P\left(\lim_{n \to \infty} \overline{X}_n = \mu\right) = 1. \tag{36}$$

This result implies if we define the sample space of all infinite sequences of experiments $(X_1, X_2, \ldots)$, then the probability of sampling a sequence whose sample mean converges to μ is 1.

We omit the proof to these theorems as they are out of the scope of this paper. However, the importance of this result is as follows: say that we have a population of interest with unknown mean μ. If we sample n elements from this population at random and measure their values in a sequence $X_1, X_2, \ldots, X_n$ then the mean of this sample, $\overline{X}_n$, will almost certainly be arbitrarily close to μ as n increases.

VIII. INTERLUDE

This concludes the exploratory part of our Integrative Exercise. In the following section we study our primary resource, Udeshi et al., 2017[6]. Our goal with this exploration was to learn about the different, and for the most part unrelated, topics on which this paper relies. This is helpful because Udeshi et el. describes an algorithm called *Aequitas*, whose goal is to improve the fairness of machine learning models. Thus, the

topics of machine learning and machine learning fairness are relevant in our discussion. As we will also see, machine learning robustness and the law of large numbers, as introduced in the sections above, serve as the mathematical background for *Aequitas*.

IX. UNDERSTANDING *AEQUITAS*

In this section we provide a discussion of *Aequitas* as presented in our primary source, "Automated Directed Fairness Testing." (Udeshi et al., 2017)[6]

As mentioned previously, there is no consensus within the field of machine learning on how to solve the issue of bias in models, especially since there can be conflicting definitions and forces pulling potential solutions. Thus, we must state which kind of fairness *Aequitas* will work with. Firstly, *Aequitas* defines fairness as individual fairness, which we will explain in detail below. At a high level though, individual fairness is the notion that we should classify people with similar characteristics but different sensitive features (e.g. race, gender, etc.) equally. *Aequitas* also believes that model bias stems from the preprocessing part of a learning algorithm, In other words, in the data collection. *Aequitas* also believes in counterfactual fairness, which means that we would like models to treat people belonging to protected categories better than what may happen in reality. Lastly, *Aequitas* works with a fairness constraint, it tries to maximize accuracy within a predetermined numerical fairness threshold.

Thus, the goals of *Aequitas*, given a potentially biased machine learning model, is to increase it individual fairness within the specified numerical threshold. The specific inputs of *Aequitas* are the model itself, its input features, and the dataset that was used to train it. The preprocessing assumption tells us that bias may underlie the training dataset as a reflection of the bias of the real world and therefore the model that was trained based on it may have learned those biases.

In particular, *Aequitas* will systematically explore the input space and find inputs that induce discrimination. *Aequitas* combines the found inputs and the original training dataset to automatically retrain the model, in the hopes of improving its fairness.

The novelty of this approach is that the retraining dataset is generated by a directed approach, in contrast to randomly selecting inputs as done by other state-of-the-art fairness algorithms. Furthermore, the directed algorithm relies on the concept of robustness of machine learning models to find discriminatory inputs for the retraining dataset. Robustness states that small changes to the input of a classifier will not change its estimated label. Even though Fawzi et al. (2015) found that there is an upper bound to robustness against adversarial inputs, these inputs are found only in small regions of the input space. *Aequitas*' directed approach will eventually avoid these regions if it fails to find discriminatory inputs.

A. Outline of the Algorithm

First, we start with a motivating example. Suppose that we count with a machine learning model that takes in the characteristics of a person, such as their education and general qualifications as well as their gender and race, and outputs a hiring recommendation. If we have two inputs to this model which have the same qualifications and requirements relevant for a job, we would like the model to recommend hiring both. However, if these inputs differ by a sensitive feature, let's say gender, and are classified differently, then the model is biased against gender. Then, we say the the inputs are discriminatory and the model violates individual fairness.

Note that in the above scenario, any difference of model classification is considered discriminatory since we only consider two options: hiring or not hiring. Sometimes we will consider other kinds of numerical outputs and in those cases we use a fairness threshold: if the difference between the outputs is within the threshold, then we do not consider the inputs discriminatory.

Henceforth, we describe an algorithm, *Aequitas*, to systematically find discriminatory inputs to retrain our machine learning model and improve individual fairness. Overall, *Aequitas* has three big steps:

a. Global search. In this step, Aequitas uniformly samples the input space and saves any discriminatory inputs that it finds. The authors of the paper claim that if there are any discriminatory inputs, then the algorithm is almost guaranteed to find at least one. Note that this step is repeated several times. Moreover, an input I is considered discriminated against if there is an input I' that is the same as I except for the value of a protected category such as gender or race. If $|f(I) - f(I')|$ is larger than a predefined fairness threshold, then I is considered a discriminatory input.

b. Local search. Given the set of discriminatory inputs from the previous step, we claim, by robustness, that in the neighborhood of each individual discriminatory input there will be more discriminatory inputs.

Below is an overview of the local search step in limited detail. The notation below will be used throughout this section.

Let $\mathbb{I}$ represent the input domain. That is, $\mathbb{I}$ is the set of all possible inputs to the model. Let $P_1, \ldots, P_n$ denote each the value of each parameter of a particular input $I \in \mathbb{I}$. Let $\mathbb{I}_k$ be the set of all values that the parameter P_k can take. Then note that $\mathbb{I} = \mathbb{I}_1 \times \cdots \times \mathbb{I}_n$ as $\mathbb{I}$ is the set of all possible inputs, and thus it is also the Cartesian product of all of the possible values of each individual parameter. An input parameter p from $P_1, \ldots, P_n$ can potentially be discriminatory if the classifier should not be biased against specific values in I_p. For example p could be gender in a creditworthiness classifier—we do not want classifiers predicting the income of a person based on gender.

Let $I \in \mathbb{I}$, let I_k be the value of the parameter P_k in I, let $I^{(d)}$ be a discriminatory input that we found on the previous part of the algorithm, and let $P_{\text{disc}} \subseteq \bigcup_{i=1}^{n} P_i$ be the set of parameters that we hypothesize might induce discrimination.

There are three flavors of Aequitas' local search:

1. Aequitas randomly chooses a parameter
$$p \in \bigcup_{i=1}^{n} P_i \backslash P_{\text{disc}}.$$
Then a small perturbation δ usually chosen at random from $\{1,-1\}$ is added to $I_p^{(d)}$.
2. The parameter p is still chosen uniformly at random, but given a perturbation δ of $I_p^{(d)}$ that consistently yields discriminatory inputs, then δ is chosen with higher probability.
3. We augment the previous approach by also picking parameters p that consistently yield discriminatory inputs with higher probability.

After we find the perturbed input, we add it to the retraining dataset if it is a discriminatory input.

The approach described works because of robustness (see Section VI). That is, we expect that slightly perturbed inputs in the neighborhood of initially found discriminatory inputs will also induce discrimination.

Moreover, by the law of large numbers, the average amount discriminatory inputs found after a large number of iterations will approximate the true ratio of discriminatory inputs in $\mathbb{I}$.

c. Retraining. We add a portion of the discriminatory inputs found to the original dataset and train a new model with it. We only add a portion of the inputs because these inputs are artificially generated so they do not "represent" reality. Since this new dataset could skew the behavior of the model, we try to add different amounts of discriminatory inputs and gauge the sensitivity of the model. Note that this reflects Counterfactual fairness: we add these discriminatory inputs, which are generated and not "factual," to the original dataset to improve the fairness of a model.

In summary, *Aequitas* begin by sampling uniformly at random from the input space of the model in question. Whenever *Aequitas* finds a discriminatory input, it adds it to the retraining set. By the Law of Large Numbers (see Section VII) there is a high probability that we will find discriminatory inputs if we sample enough times. We look at the neighborhood of each input in previous step to find more discriminatory inputs. We do this by perturbing the inputs, where the perturbations are probabilistic and learn from previous iterations. After this is done, we use the discriminatory inputs found to retrain the model.

B. Aequitas, formal declaration

Now we proceed with the formal declaration of the algorithm. The discussion in the previous discussion is detailed enough for a superficial understanding of the algorithm. Here we concern ourselves with the mathematical definitions and justifications of *Aequitas*. Note that we use the notation that we used in the previous section.

We start by giving more notation and definitions that we will need.

Let $P = \bigcup_{i=1}^{n} P_i$ be the set of parameters of inputs in $\mathbb{I}$.

Definition IX.1 (Individual Fairness, Discriminatory Input)**.** Let f be a classifier under test, and let γ be a user defined discrimination threshold. Let $I \in \mathbb{I}$. Suppose there is $I' \in \mathbb{I}$ such that there is a non-empty subset $Q \subseteq P_{\text{disc}}$ and such that for every $q \in Q$, $I_q \neq I'_q$ and for all $p \in P\backslash Q$, $I_q = I'_q$. If $|f(I) - f(I')| > \gamma$ then I is a discriminatory input of the classifier f and reflects the violation of Individual Fairness in the classifier f.

A perturbation g is a function $g : \mathbb{I} \times (P\backslash P_{\text{disc}}) \times \Gamma \to \mathbb{I}$ where $\Gamma = \{+1,-1\}$ defines the directions in which we can perturb an $I \in \mathbb{I}$. Then $g(I,p,\delta)$ is the input $I' \in \mathbb{I}$ such that $I'_p = I_p + \delta$ and such that for all $q \in P\backslash\{p\}$ we have that $I'_q = I_q$.

Now we formally define *Aequitas*.

a. Global search. (See Algorithm 1) As mentioned before, we want to take a subset of discriminatory inputs from $\mathbb{I}$ to drive our local search algorithm. To this end, we select an input $I \in \mathbb{I}$ at random, and generate a set of inputs $\mathbb{I}^{(d)}$ from I that cover all possible values of P_{disc}. Finally, from $\mathbb{I}^{(d)}$ we find discriminatory inputs as defined previously.

Algorithm 1: Global Search

```
1  procedure GLOBAL_EXP(P, P_disc)
2      disc_inputs ← ∅
3      Let P' = P\P_disc
       // N is the number of trials
4      foreach i in (0,N) do
5          Select I ∈ 𝕀 at random
           // 𝕀^(d) extends I with all possible values
               of P_disc
6          𝕀^(d) ← {I' | ∀p ∈ P', I_p = I'_p}
7          if ∃I', I'' ∈ 𝕀^(d), |f(I') − f(I'')| > γ then
8              disc_inputs ← disc_inputs ∪ {I}
9          end
10     end
11     return disc_inputs
12 end
```

b. Local search. (See Algorithm 2) Let `disc_inputs` be the set of discriminatory inputs found on the previous step. Now we want to find discriminatory inputs in the neighborhood of inputs in `disc_inputs`. Remember that we can do this because of the robustness of machine learning models.

Thus, Aequitas perturbs an input $I \in$ `disc_inputs` by changing the value of I_p by $\delta \in \{+1,-1\}$ where $p \in P\backslash P_{\text{disc}}$. Note that by changing I_p, I is now a different input, and it will be further perturbed in the inner loop of Algorithm 2.

Now, the perturbations on the inputs are chosen at random. Perturbations are defined by picking a feature to perturb and a direction or value by which to perturb so we randomly pick both values separately. In this regard, let σ_{pr} be the array such that $\sigma_{\text{pr}}[p]$ is the probability of picking p for any $p \in P\backslash P_{\text{disc}}$. The probability of picking a value of perturbation will depend on which feature we choose to perturb, so let σ_v be the array such that $\sigma_v[p]$ is the chance of picking a perturbing the feature denoted by p by $\delta = -1$ for any $p \in P\backslash P_{\text{disc}}$.

Algorithm 2: Local Search

```
1  procedure LOCAL_EXP(disc_inputs, P, P_disc, Δ_υ, Δ_pr)
2      Test ← ∅
3      Let P' = P\P_disc
4      Let σ_pr[p] = 1/|P'| for all p ∈ P'
5      Let σ_υ[p] = 0.5 for all p ∈ P'
6      foreach I ∈ disc_inputs do
           // N is the number of trials
7          foreach i in (0,N) do
8              Select p ∈ P' with probability σ_pr[p]
9              Select δ = −1 with probability σ_υ[p]
               // I is modified as a side effect of
                  modifying I_p
10             I_p ← I_p + δ
               // 𝕀^(d) extends I with all possible
                  values of P_disc
11             𝕀^(d) ← {I' | ∀p ∈ P', I_p = I'_p}
12             if ∃I', I'' ∈ 𝕀^(d), |f(I') − f(I'')| > γ then
                   // Add perturbed input I
13                 Test ← Test ∪ {I}
14             end
15         end
16         update_prob(I, p, Test, δ, Δ_υ, Δ_pr)
17     end
18     return Test
19 end
```

As noted before, the parameter p is chosen with probability $\sigma_{\text{pr}}[p]$ where initially $\sigma_{\text{pr}}[p] = \frac{1}{|P \backslash P_{\text{disc}}|}$. Once p is chosen, we need to choose $\delta \in \{+1, -1\}$ where the probability of $\delta = +1$ is $\sigma_{\upsilon}[p]$ and $\delta = -1$ is $1 - \sigma_{\upsilon}[p]$. Initially, $\sigma_{\upsilon}[p] = 0.5$ for all $p \in P \backslash P_{\text{disc}}$.

There are three kinds of Aequitas based on how we update $\sigma_{\text{pr}}[p]$ and $\sigma_{\upsilon}[p]$.

Aequitas Random. This is when $\sigma_{\text{pr}}[p]$ and $\sigma_{\upsilon}[p]$ are not updated. This is equivalent to sampling inputs uniformly at random from the neighborhood of an input $I \in$ `disc_inputs`.

Algorithm 3: Aequitas semi-directed update probability

```
1  procedure UPDATE_PROB(I, p, Test, δ, Δ_υ, Δ_pr)
2      if (I ∈ Test ∧ δ = −1) ∨ (I ∉ Test ∧ δ = +1) then
3          σ_υ[p] ← min(σ_υ[p] + Δ_υ, 1)
4      end
5      if (I ∉ Test ∧ δ = −1) ∨ (I ∈ Test ∧ δ = +1) then
6          σ_υ[p] ← max(σ_υ[p] − Δ_υ, 0)
7      end
8  end
```

Aequitas semi-directed. (See Algorithm 3) In this case, we drive the test generated by updating σ_{υ}. Note that p is still chosen uniformly at random. For each $p \in P \backslash P_{\text{disc}}$ we initialize the probability of the perturbation value $\delta = -1$ as $\sigma_{\upsilon}[p]$ and $\delta = +1$ as $1 - \sigma_{\upsilon}[p]$. Based on whether a particular δ led to a discriminatory input, $\sigma_{\upsilon}[p]$ is increased or decreased by a user determined value Δ_{υ}.

Aequitas fully-directed. (See Algorithm 4) This approach extends Aequitas semi-directed by systematically updating $\sigma_{\text{pr}}[p]$. For any $p \in P \backslash P_{\text{disc}}$ we initialize $\sigma_{\text{pr}}[p] = \frac{1}{|P \backslash P_{\text{disc}}|}$. If the perturbation of p by δ (which is chosen as noted in Aequitas semi-directed) leads to a discriminatory input then we add a user determined value Δ_{pr} to $\sigma_{\text{pr}}[p]$. To reflect this change in probability we normalize $\sigma_{\text{pr}}[p'] = \frac{\sigma_{\text{pr}}[p']}{\sum_{x \in P \backslash P_{\text{disc}}} \sigma_{\text{pr}}[x]}$ for every $p' \in P \backslash P_{\text{disc}}$. We do this because we still need the probabilities to add up to 1 even after updating $\sigma_{\text{pr}}[p]$.

Algorithm 4: Aequitas fully-directed update probability

```
1  procedure UPDATE_PROB(I, p, Test, δ, Δ_υ, Δ_pr)
2      if (I ∈ Test ∧ δ = −1) ∨ (I ∉ Test ∧ δ = +1) then
3          σ_υ[p] ← min(σ_υ[p] + Δ_υ, 1)
4      end
5      if (I ∉ Test ∧ δ = −1) ∨ (I ∈ Test ∧ δ = +1) then
6          σ_υ[p] ← max(σ_υ[p] − Δ_υ, 0)
7      end
8      if I ∈ Test then
9          σ_pr[p] ← σ_pr[p] + Δ_pr
10         σ_pr[q] ← σ_pr[q] / (Σ_{x∈P\P_disc} σ_pr[x]) for all q ∈ P\P_disc
11     end
12 end
```

c. Estimation using the Law of Large Numbers. Using Aequitas we can estimate the percentage of discriminatory inputs in $\mathbb{I}$. Let $X_1, X_2, \ldots X_K$ be a sequence of random variables, each counting the proportion of discriminatory inputs in a random sample of m inputs from the input space. Note that $E[X_i] = m^*$ for all $1 \leq i \leq K$, where m^* is the true proportion of discriminatory inputs in the input space. Then by the law of large numbers, the average of the samples, $\overline{X} = K^{-1} \sum_{i=1}^{K} X_i$, will tend to be close to the expected value as K increases. Hence, $\overline{X} \to m^*$ as $K \to \infty$. Whence, we call `LLN_Fairness_Estimation` the function that takes in a model and samples its input space uniformly at random multiple times to find discriminatory inputs. If we do this enough times, we will get a "good" estimation of the true ratio of discriminatory inputs in the input space. For a usage of this function, see Algorithm 5, line 14.

d. Automatic retraining of the model. (See Algorithm 5) Let `Test` be the set of generated test inputs from the local search step. We use `Test`—whose elements show the violation of the desired properties that we want our model to have—to retrain our machine learning model. The strategy we use is to add portions of `Test` to the original training dataset. This is because adding all of the elements in `Test` to the training dataset may skew the model, as the elements in `Test` may be an over represented region in the input space.

Now we describe how we select portions of `Test` to add to the training dataset. Suppose that $M = |\texttt{Test}|$. Then for $i \in [2, 7]$, we choose p_i randomly in a range between $[2^{i-2}, 2^{i-1}]$ and select $\frac{M * p_i}{100}$ elements from `Test` at ran-

dom. Intuitively, p_i represents the percentage of elements from `Test` that we will select. After all of the iterations, we keep the retrained model that had the least percentage of discriminatory inputs in its input space, as computed by `LLN_Fairness_Estimation`.

The intuition behind picking p_i from an exponentially increasing range $[2^{i-2}, 2^{i-1}]$ is to maintain our level of fairness relatively high while also making the execution of the program fast.

Algorithm 5: Retraining

```
procedure Retraining(f, Test, training_data)
    N ← ∞
    f_cur ← f
    foreach i in [2,7] do
        p_i ← a real number between [2^{i-2}, 2^{i-1}]
        if p_i > 100 then
            Exit the loop
        end
        k ← len(training_data)
        n_addn ← (p_i · k) / 100
        TD_addn ← n_addn randomly selected inputs from
          training_data
        TD_new ← TD_addn ∪ training_data
        f_new ← model trained using TD_new
        // LLN_Fairness_Estimation estimates the
           proportion of discriminatory inputs
           using the law of large numbers
        fair_cur ← LLN_Fairness_Estimation(f_cur)
        fair_new ← LLN_Fairness_Estimation(f_new)
        if fair_cur > fair_new then
            f_cur ← f_new
        else
            Exit the loop
        end
    end
    return f_cur
end
```

C. Conclusion

We have described *Aequitas* in its entirety and given a brief survey on the theoretical topics on which it relies. In particular, we discussed the issue that arises when one wants to define fairness and how to solve that specific kind of fairness. Each one of these solutions is different and at times even contradictory. In this way, *Aequitas* is opinionated: it attempts to improve a model's Individual Fairness and it assumes that the bias that happens comes from the training dataset itself.

We can see how these two forces, Preprocessing and Individual Fairness, play a role in the statement of *Aequitas*: we define discriminatory inputs in terms of Individual Fairness and retrain the model by adding a set of them to the training dataset.

Within this realm of machine learning fairness algorithms, *Aequitas* is special because the retraining dataset, the one containing all of the discriminatory inputs, is not generated completely at random. A portion of the dataset, specifically the one that is found in the Local Search step, is found in a directed way: *Aequitas* looks for discriminatory inputs in the neighborhood of other discriminatory inputs.

Of course, there are limitations to *Aequitas*. In Udeshi et. al. there is no elaboration in the retraining techniques that *Aequitas* uses and they justify this by claiming that the essence of *Aequitas* is automated training dataset generation. On the other hand, the formal statement of *Aequitas* in the primary source assumes integer-valued features in the model's inputs, whereas the discussion on this article assumed that the inputs could be continuous.

Another shortcoming to *Aequitas* is its dependence on robustness. Recall that the Local Search step of the algorithm assumes that the neighborhood of discriminatory inputs will behave similarly, i.e. we will be able to find more discriminatory inputs. We found the citations from Udeshi et. al. on this topic not satisfactory and we question whether the notion of robustness is justified correctly.

All in all, fairness remains an important topic in machine learning as this technologies become more and more ubiquitous in our daily lives. We hope that *Aequitas* can be applied in real world scenarios and moreover that these conversations around fairness stay relevant.

Lastly, Udeshi et. al. provided a Python proof-of-concept implementation of their algorithm. The next section, which we have decided to include sort of as an appendix, explores the code in some detail.

Appendix A: Understanding *Aequitas* Code

This section will highlight some important details of the Aequitas implementation. We begin by introducing the modules that were used to run Aequitas, after which we look at the core snippets of the code, comparing it to the intentions of the original algorithm.

1. Modules Used

```
sklearn
```

Sklearn (Scikit Learn) module contains several machine learning algorithms that can be used to train models, including SVM, Ensemble, Decision Tree, and Neural Networks.

```
numpy
```

Numpy (Numerical Python) is a module that is frequently used in machine learning to manipulate the input and output and perform matrix operations during training. In Aequitas, Numpy plays an integral part in evaluating whether an arbitrary input is discriminatory.

```
scipy
```

Scipy (Scientific Python) is also used frequently during machine learning training. In the Aequitas code, scipy's "basin hopping" algorithm is used during local perturbation to discover more discriminatory inputs.

2. Structure of the Code

Aequitas parameters are set up using a configuration file. Parameters defined in the configuration file persist throughout the Aequitas code.

```
params = 13

sensitive_param = 9 # Starts at 1.

input_bounds = []
input_bounds.append([1, 9])
input_bounds.append([0, 7])
...
input_bounds.append([0, 39])

classifier_name =
    'Decision_tree_standard_unfair.pkl'

threshold = 0

perturbation_unit = 1

retraining_inputs = 'Retrain_Example_File.txt'
```

Listing 1. config.py

`params` refers to the number of parameters in the dataset. `sensitive_param` is the index (based on 1) of the sensitive text (i.e. gender). `input_bounds` are the domain, or the value range of each of the parameters. `classifier_name` is the name of the file containing the sklearn-trained classifier to test fairness of. `threshold` is the discrimination threshold. `perturbation_unit` is the unit by which Aequitas perturbs the input in the local search. `retraining_inputs` is the name of the dataset used for the retraining. This dataset is the result of Aequitas algorithm execution.

3. Code Snippets

Now we look at the code snippets that are integral to each part of the Aequitas algorithm.

a. Training a Model

In the initial training of a model, X and Y, which are input features and prediction, respectively, are extracted from the input dataset, and are used to train a model such as a Decision Tree Classifier.

```
X = np.array(X)
Y = np.array(Y)
model = DecisionTreeClassifier()
model.fit(X, Y)
```

Listing 2. Generate_Sklearn_Classifier.py

b. Evaluating Fairness

Throughout Aequitas, evaluating whether a given input is 'biased' plays an integral role in determining 1) how fair the model is, and 2) what input gets added to the new retraining dataset. The series of code below illustrates this procedure.

```
def evaluate_input(inp):
    inp0 = [int(i) for i in inp]
    inp1 = [int(i) for i in inp]
```

Listing 3. Sklearn_Estimation.py

In the original code, it is hard coded that the sensitive feature (i.e. gender) must be binary, meaning only two values are possible. This is why the original input *inp* is cloned twice. If the sensitive feature is non-binary, the number of clones will match the number of possible values in the sensitive feature's domain.

Once there are two clones, only the value of the sensitive feature is changed so that each one has one of the two possible values, in this case 0 and 1.

```
    inp0[sensitive_param - 1] = 0
    inp1[sensitive_param - 1] = 1
```

Then, the model predicts the outcomes (y) of the two clones, and if the outcomes differ more than the threshold, we deem *inp* discriminatory.

```
    out0 = model.predict(inp0)
    out1 = model.predict(inp1)
    return abs(out0 - out1) > threshold
```

c. Discriminatory Input Search

The aforementioned algorithm is used both in the *global* search and *local* search to detect discriminatory inputs. Aequitas first performs global search, where discriminatory inputs are found by sampling uniformly at random from the input space. Scipy's *basinhopping*, an algorithm similar to simulated annealing, is the algorithm that is used for this purpose. Simulated Annealing is an algorithm that takes into account both optimized direction of search and randomness to reach a goal (in this case, finding more discriminatory inputs). When global search finishes, local search uses *basinhopping* again, to find more discriminatory inputs in the neighborhood of these aforementioned inputs.

```
basinhopping(evaluate_global, initial_input,
    stepsize=1.0, take_step=global_discovery,
    minimizer_kwargs=minimizer,
            niter=global_iteration_limit)
```

Listing 4. Global Discovery

The parameter `take_step` takes in a function that performs the random displacement of features within the input space. In our case, that function is `Global_Discovery`, which given an input, randomly modifies the values of each of the parameters within the respective ranges, returning a new input sourced from the input space. This random sampling characterizes global search. When this new input is created, whether or not it is a discriminatory put is determined using the `evaluate_global` function, which employs a similar mechanism of evaluating an input as one that was introduced earlier. The only difference is that this time the discriminatory inputs are saved in an array.

For local search, there are three approaches. The difference between these approaches is whether the probability of choosing the perturbation direction (+1, -1) and/or choosing a feature to perturb are modified throughout the algorithm. In Aequitas Random, the feature to perturb and perturbation direction are chosen uniformly at random and not in a directed way. It represents a random exploration of the neighborhood of discriminatory inputs discovered through global search.

```
feature = random.randint(0, 12)
direction = [-1, +1]
# perturbation
x[feature] = x[feature] + random.choice(direction)
```

Listing 5. Local Discovery-Aequitas Random

In Aequitas Semi-Directed, the feature to perturb is still chosen randomly, but the direction of perturbation is now determined by the *direction_probability* for that specific feature.

```
init_prob = 0.5
direction_probability = [init_prob] * params
```

Direction of -1 is chosen with the probability of `direction_probability[feature]` and direction of +1 is chosen with the probability of `1 - direction_probability[feature]`. At first, the *direction_probability* of each input feature is the same, and choice of direction is random, as it was in Aequitas Random.

```
direction_choice = np.random.choice(direction,
    p=[direction_probability[feature], (1 -
    direction_probability[feature])])

x[feature] = x[feature] + (direction_choice *
    perturbation_unit)
```

However, in Aequitas Semi-Directed, the perturbed input is evaluated, and based on its "biasedness", *direction_probability* is adjusted.

```
ei = evaluate_input(x)

if (ei and direction_choice == -1) or (not ei and
    direction_choice == 1):
    direction_probability[feature] = min(
        direction_probability[feature] +
            (direction_probability_change_size *
            perturbation_unit), 1)

elif (not ei and direction_choice == -1) or (ei and
    direction_choice == 1):
    direction_probability[feature] = max(
        direction_probability[feature] -
            (direction_probability_change_size *
            perturbation_unit), 0)
```

Put simply, if the perturbed input is discriminatory, the *direction_probability* of the `direction_choice` will be rewarded because by robustness, this must mean that neighboring inputs in this direction will also likely be discriminatory. In other words, the algorithm increases `direction_probability[direction_choice]` to incentivize perturbing in this direction. Conversely, if the perturbed input is not discriminatory, `direction_probability[direction_choice]` will be decreased to disincentivize perturbation in this direction.

In Aequitas Fully-Directed, both the direction of perturbation and the feature to perturb are chosen with differing probabilities. The algorithm learns which features are more associated (correlated) with the sensitive feature, (i.e. frequency of parental leave and gender) and attempts to find discriminatory inputs more efficiently by adjusting those parameters with greater probability.

First, *direction_probability* and *param_probability* are

initialized to be equal across all features.

```
init_prob = 0.5
direction_probability = [init_prob] * params
direction_probability_change_size = 0.001
param_probability = [1.0/params] * params
param_probability_change_size = 0.001
```

The very first input selection is random, as the probabilities are the same. However, after the initial input is evaluated, both *direction_probability* and *param_probability* are adjusted accordingly, in a manner similar to Aequitas Semi-Directed, in that the algorithm incentivizes the selection of feature and direction that yields a discriminatory input. For brevity I only include how *param_probability* is adjusted, since the adjustment to *direction_probability* have been discussed previously.

```
x[param_choice] = x[param_choice] +
    (direction_choice * perturbation_unit)

ei = evaluate_input(x) #True if biased, False
    otherwise

if ei:
    param_probability[param_choice] =
        param_probability[param_choice] +
        param_probability_change_size
    normalise_probability()
else:
    param_probability[param_choice] =
        max(param_probability[param_choice] -
        param_probability_change_size, 0)
    normalise_probability()
```

d. *Retraining*

Instead of naiively adding all the discriminatory inputs into the training set which might result in a skewed representation of the overall data, Aequitas chooses only a subset of the generated inputs to add at each iteration, checking if training with the additional inputs made fairness better or worse. If the newly trained model has better fairness, keep adding more inputs and see if fairness can be further improved. If the newly trained model yields worse fairness, return the current model without updating.

The additive percentage of the generated inputs (how much of it we add) is chosen randomly between

$$[2^i, 2^{i+1}]$$

where i is the 0-based index of iteration.

```
additive_percentage = random.uniform(pow(2, i),
    pow(2, i + 1))
num_inputs_for_retrain = int((additive_percentage *
    len(X))/100)
```

This means, that the more iteration you go through of this process, more of the remaining generated inputs we add to the training set. Of course, if the percentage becomes greater than a 100, we exit. The code below illustrates the retraining procedure.

After the additive percentage has been decided, Aequitas sources randomly from the retraining dataset and retrains.

```
retrained_model = retrain(X_original, Y_original,
    np.array(X_additional), np.array(Y_additional)
```

It is important to learn about the code implementations of Aequitas not only to appreciate the intricacies of realizing the algorithm but also to detect where it might deviate from the original algorithm. For example, the original algorithm enables non-binary sensitive features to be processed, but the code implementation was limited to binary sensitive features. From this understanding, we were able to figure out how we might improve the code to stay truer to the original intentions of the algorithm.

ACKNOWLEDGMENTS

We would like to thank Professor Anna Rafferty for the conversation on machine learning fairness, and Professor Dave Musicant for the discussion about support vector machines.

REFERENCES

[1] P. Olson, "The Algorithm That Beats Your Bank Manager," (2011).
[2] E. Reicin, "AI Can Be A Force For Good In Recruiting And Hiring New Employees," (2021).
[3] K. Ferral, "Wisconsin Supreme Court allows state to continue using computer program to assist in sentencing," (2016).
[4] J. Angwin, J. Larson, S. Mattu, and L. Kirchner, "Machine Bias: There's software used across the country to predict future criminals. And it's biased against blacks." (2016).
[5] J. Dastin, "Amazon scraps secret AI recruiting tool that showed bias against women," (2018).
[6] S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering , 98–108 (2018).
[7] S. Russell and P. Norvig, *Artificial Intelligence: a Modern Approach*, 4th ed. (Pearson, 2020).
[8] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: testing software for discrimination," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (ACM, New York, NY, USA, 2017).
[9] M. B. Zafar, I. Valera, M. G. Rodriguez, and K. P. Gummadi, "Fairness Constraints: Mechanisms for Fair Classification," (2015).
[10] B. Hutchinson and M. Mitchell, "50 Years of Test (Un)fairness: Lessons for machine learning," in *FAT* 2019 - Proceedings of the 2019 Conference on Fairness, Accountability, and Transparency* (Association for Computing Machinery, Inc, 2019) pp. 49–58.
[11] A. Fawzi, O. Fawzi, and P. Frossard, "Analysis of classifiers' robustness to adversarial perturbations," (2015).
[12] R. P. Dobrow, *Probability with applications in R* (2013).

Experimentation with *Aequitas*: A Report

Yemi Shin, Yunping Wang, Michael Worrell, and Juanito Zhang Yang
Department of Computer Science, Carleton College, 300 North College Street, Northfield, MN, 55057

(Dated: March 16, 2022)

I. INTRODUCTION

This report will give a summary of the set of experiments we conducted on Aequitas using various datasets. The purpose of the experimentation was to 1) get familiar with the Aequitas code and 2) to test the limits of Aequitas (by perhaps attempting to break the code) and 3) to subsequently spot possible areas of improvement.

II. EXPERIMENTATION WITH EMPLOYEE DATASET

A. Objective

The purpose of this experimentation was to successfully run Aequitas on an arbitrary data set. The employee data set was sourced from Kaggle[1] and was chosen because it had a column for 'gender' which is the only type of sensitive feature that Aequitas can currently assess. In the process of making Aequitas work I refactored the codebase and made it generalizable to any type of data set given in a csv format.

B. Methods

Prior to testing, the codebase was modified so that only the `config.py` had to be modified in order to make Aequitas work for a particular data set.

The employee data set had 9 columns: education, year of joining the company, city, payment tier, age, gender, ever benched, experience in current domain, and whether or not the employee left the company in less than two years. The column to be predicted was whether or not the employee would leave the company in less than two years. The data set had 4654 data points.

The data set was preprocessed to numerically encode categorical values and to denote 'yes' and 'no' binaries with '1', '-1' instead of '1', '0'. This step was important, for discriminatory input evaluation depended on the absolute value of the difference between outputs being either equal to 0 or greater.

After training a 'DecisionTreeClassifier' on the preprocessed dataset, the resulting model was subjected to Aequitas Random, Aequitas Semi-Directed, and Aequitas Fully-Directed search in order to find discriminatory inputs.

After every search, the original model was retrained with the collected discriminatory inputs to see if the model's fairness could be improved.

C. Results

Below are the results of the searches.

- With *Aequitas Random*, percentage of discriminatory inputs found of all tested inputs was 36%. The number of discriminatory inputs it found were 20643.
- With *Aequitas Semi-Directed*, percentage of discriminatory inputs found of all tested inputs was 49%. The number of discriminatory inputs found were 8579.
- With *Aequitas Fully-Directed*, percentage of discriminatory inputs found of all tested inputs was 42%. The number of discriminatory inputs found were 3414.

Below are the results of retraining the original model with the discriminatory inputs collected from each of the aforementioned Aequitas search strategies.

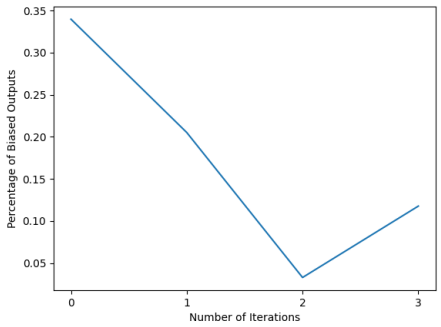

Figure 1. Retraining with Uniformly(Randomly) Derived Discriminatory Inputs

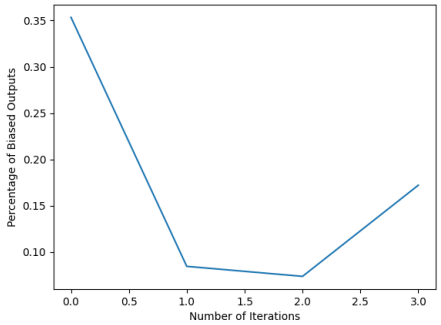

Figure 2. Retraining with Semi-Directed Derived Discriminatory Inputs

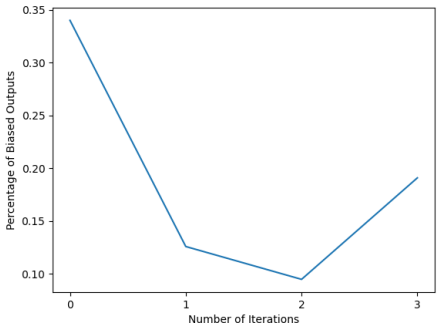

Figure 3. Retraining with Fully-Directed Derived Discriminatory Inputs

D. Discussion

The difference between the three Aequitas search strategies is not so much the raw number of discriminatory inputs gathered as it is the efficiency of collection. Random search, although in raw numbers found the most discriminatory inputs, is difficult to justify, for 20643 is nearly five times the size of the original data set. This number, however, is still only 36% of the entire tested inputs, which means the total number of inputs that were tested during the search was far greater than reasonable. In the end, Aequitas Random is highly inefficient compared to the other two methods which are more directed. For example, Aequitas Fully-Directed found the least number of training inputs, but in terms of efficiency of data gathering, it reigned superior over the other two because it tried the fewest number of inputs and yet yielded a high percentage of the desired inputs.

Additionally, the different search strategies did not yield different 'quality' of retraining inputs. That is, a more directed Aequitas search did not necessarily yield inputs that would improve the model *better* than ones that were randomly generated. As seen in Figure 1 through Figure 3, there seems to be no significant correlation between fairness improvement and search strategy. This is likely because the only difference between the three, as mentioned before, is the efficiency with which the inputs are found, not the quality of the inputs.

One thing to note is that Aequitas, on average, took 7-13 minutes to run on a data set of size 4000, which is undesirable if our goal is to make this a scalable web application. We can improve the overall performance by parallelizing the local search process using a multiprocessing module.

III. EXPERIMENTATION WITH BANKNOTE AUTHENTICATION DATASET AND EXPERIMENTATION WITH HABERMAN'S SURVIVAL DATA

A. Objective

The purpose of this experimentation was multi-fold. First, this experiment was performed to check whether Aequitas was able to run when provided a non-gender attribute as a sensitive attribute. Second, we wanted to check whether Aequitas would identify discriminatory input in a data set where the output *should* be reliant on the sensitive attribute, and if Aequitas would increase model fairness by retraining the dataset if this was the case.

The Banknote Authentication[2] and Haberman Survival[3] data sets were sourced from the UC Irvine Machine Learning Repository. The Banknote Authentication data set was chosen because of non-integer parameter values, as well as its apparent lack of attributes that could be considered unfairly discriminatory. The Haberman Survival data set was chosen for its use of integer parameter values, while also appearing to lack attributes that could be considered unfairly discriminatory.

B. Methods

The banknote authentication data set had 5 columns: variance of Wavelet Transformed image, skewness of Wavelet Transformed image, curtosis of Wavelet Transformed image, entropy of image, and whether or not the banknote present in the image was genuine or forged. The column to be predicted was whether or not the banknote in the image was forged. This data set had 1372 data points.

The Haberman survival data set had 4 columns: the age of the patient at time of operation, the the year of the patient's operation (with the century excluded), the number of positive axillary nodes detected, and whether or not the patient survived 5 years or longer (survival status). The column to be predicted was the survival status. This dataset had 306 data points.

Both data sets were preprocessed in order to numerically encode categorical values and to denote 'yes' and 'no' binaries with '1', '-1' instead of '1', '0' (in the case of the banknote authentication dataset) or '1', '2' (in the case of the Haberman survival dataset). Without this step, Aequitas would not be able to treat that parameter as binary.

After training a 'DecisionTreeClassifier' on the preprocessed dataset, the resulting model was subjected to Aequitas Random, Aequitas Semi-Directed, and Aequitas Fully-Directed search in order to find discriminatory inputs.

C. Results

This result sections will be split into two sections: one discussing the results of our experimentation on the banknote authentication data set, and another discussing the results of our experimentation on the Haberman survival data set.

Banknote Dataset Results
Below are the results of the searches in the banknote authentication dataset.

- With *Aequitas Random*, percentage of discriminatory inputs found of all tested inputs was 0%. The number of

discriminatory inputs it found was 0, out of 1001 total inputs.

- With *Aequitas Semi-Directed*, percentage of discriminatory inputs found of all tested inputs was 0%. The number of discriminatory inputs found was 0, out of 1001 total inputs.

- With *Aequitas Fully-Directed*, percentage of discriminatory inputs found of all tested inputs was 0%. The number of discriminatory inputs found was 0, out of 1001 total inputs.

Because Aequitas did not produce any discriminatory input for this dataset, we were unable to retrain the original model with new data.

Haberman Survival Dataset
Below are the results of the searches in the Haberman survival dataset.

- With *Aequitas Random*, percentage of discriminatory inputs found of all tested inputs was 32.14%. The number of discriminatory inputs it found was 189, out of a total 588 total inputs.

- With *Aequitas Semi-Directed*, percentage of discriminatory inputs found of all tested inputs was 33.21%. The number of discriminatory inputs found was 186, out of 560 total inputs.

- With *Aequitas Fully-Directed*, percentage of discriminatory inputs found of all tested inputs was 32.02%. The number of discriminatory inputs found was 187, out of 584 total inputs

Below are the results of retraining the original model with the discriminatory inputs collected from each of the aforementioned Aequitas search strategies on the Haberman survival dataset.

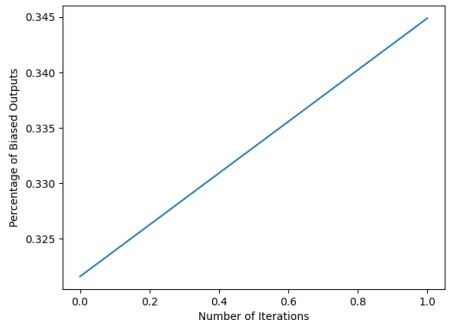

Figure 4. Retraining with Uniformly(Randomly) Derived Discriminatory Inputs

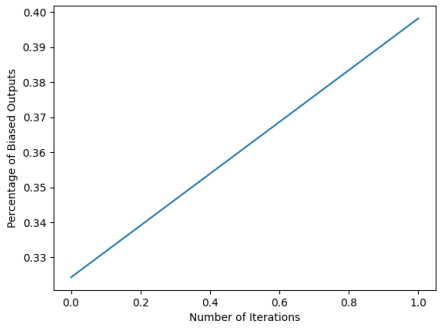

Figure 5. Retraining with Semi-Directed Derived Discriminatory Inputs

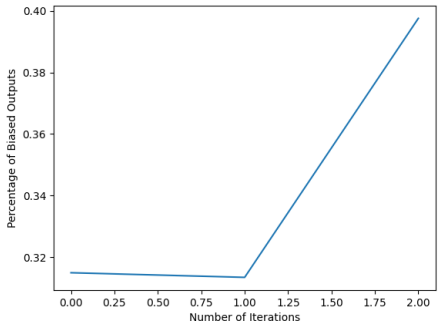

Figure 6. Retraining with Fully-Directed Derived Discriminatory Inputs

D. Discussion

Banknote Authentication Dataset

The three Aequitas search strategies failed to identify any discriminatory inputs from the banknote authentication data set.

One possible explanation for why running Aequitas on the banknote authentication data set did not identify any discriminatory inputs could be the fact that every column except for the binary classification column used non-integer real numbers, including the parameter we chose as the sensitive attribute. Our python code that automatically encoded the data converted floats into distinct integers as a side effect. This could have prevented similar values in decimal form from being identified as close in that new integer form. More experimentation is needed to see if this is the cause for the lack of discriminatory input being identified in this experiment.

Haberman Survival Dataset

The three Aequitas search strategies did not differ greatly in the amount of discriminatory input identified, ranging between 32% and 33%. The running time did not differ greatly between the three search strategies, since they each gathered the same number of inputs, ±5%.

Despite the high percentage of discriminatory inputs found, out of the three different search strategies, only Aequitas Fully-Directed was able to improve the original model by retraining using these inputs (Figure 4 through Figure 6). However, the percentage of biased outputs decreased by so little (<1%) that this is likely not a result of those inputs being truly discriminatory. It is possibly a result of overfitting to this particular data set. Aside from that iteration of Aequitas Fully-Directed, every other attempt to retrain the model using found discriminatory inputs *increased* the percentage of biased data between 2 percentage points and 8 percentage points.

These results are promising for the following reason. The sensitive attribute we selected (the number of positive axillary nodes detected) *should* be a deciding factor in determining the column to be predicted (Whether the patient survives 5 or more years following their operation). When we configured Aequitas to assume that the number of positive axillary nodes detected should not impact the survival rate of a patient, the model was retrained using data *that did not reflect the real world*. Since Aequitas did not end up notably improving the original model using this assumption that does not reflect the real world, it is an indication that Aequitas will be resistent to overtly faulty configuration.

IV. EXPERIMENTATION WITH STUDENT PERFORMANCE DATASET

A. Objective

The purposes of this experiment was to test the scalability of the fairness estimation process. *Aequitas* measures the fairness of a data set by randomly sampling with replacement from the data set and measuring the fairness by counting all the examples that has the same classification in all other attributes, but different classifications for the sensitive attribute. It then computes the percent bias by computing the percentage of the biased input in the entire data set. In this experiment, we aimed to test the effect of changing the sample size on the error rate of the fairness testing. It is important to know the relative accuracy of fairness estimation, since we are evaluating the success of the program based on how much it increases fairness. If we increase the sample size, we would in theory get a more accurate result. However, it would also require more time.

B. Methodology

Since we are sampling with replacement, this method would never produce a "true" measurement given that our data set is large enough for every random sampling process to produce a different set of samples. However, we hypothesized that the more we sample, the more our results would approach a "true value." We can justify this by citing the central limit theorem. We know that as we increase the sample size, the probability distribution of the bias in the data set will be a Gaussian curve. This allows us to treat the peak as the "true value." However, this is improbable as we don't have the time or the computational power to sample indefinitely. Thus, we need to determine a sample size that will create sufficiently small fluctuation to allow us to deem the peak as being close enough to the "true value."

It was important to determine a threshold for the error value we could call a "small fluctuation." Since the final bias could be interpreted as a Gaussian curve, we needed to choose an error that translated to a close cluster around the Gaussian peak. A common width measurement for Gaussian is to measure the width of the Gaussian at height that is $1/e$ of the total height. All value should have an 86% chance of being within this interval, which means that they would have 14% chance of being outside this interval. If we divide this by 2, we get 7%. Therefore, if the uncertainty is less than 7%, we would decide that our values are sufficiently close to the peak.

We designed our experiment around this principle. In particular, we measured the bias of a particular sample size by taking an average of five independent measurements. Then we computed the error by computing the standard deviation of the mean. We incremented this number by 1% of the population. We stopped once three consecutive error measurements were below the 7% threshold.

For the experiment, we chose a fictional data set on education with 1000 data points. We chose gender as the sensitive attribute.

C. Results

Figure 1 shows the plot of percent bias found versus the sample size. Error bar displayed were computed based on the standard deviation of the mean.

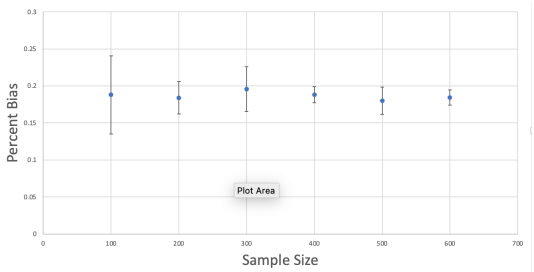

Figure 1. The plot of percent bias found versus the sample size. Error bar displayed were computed based on the standard deviation of the mean.

Figure 2 is the plot of percent error found versus the sample size. We stopped taking data after the percent error was consistently below 7%.

D. Discussion

For a population of 1000, we reached a sufficient sample size of 600, or 60%. Thus, we can use this result when we test other data sets by starting with a sample size of 60% of the original, and incrementing or decrementing as we see fit.

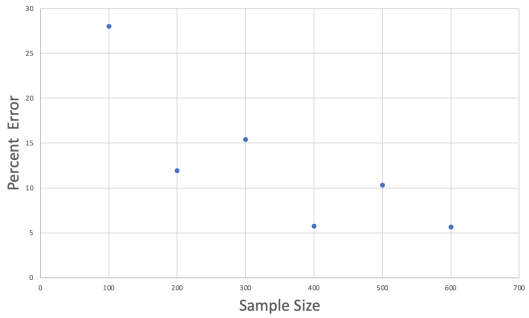

Figure 2. The plot of percent error found versus the sample size. The error was computed from the standard deviation of the mean.

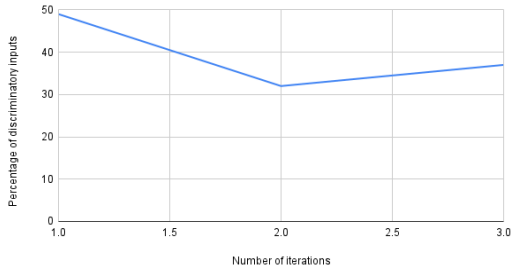

Figure 3. The plot of percent bias found versus the number of iterations.

V. EXPERIMENTATION WITH DEFAULT OF CREDIT CARD CLIENTS DATASET

A. Objective

We wanted to apply Aequitas to a data set with many more columns than the ones we have analyzed so far. For that reason, we picked a data set[4] that compiled data about the default of credit card clients. In particular, the data set described the profile of a client, and the payment and debt history of their credit card, and whether or not they had to default on their debt. In this case, the data set had 24 attributes, including the classification that we wanted to predict.

B. Methods

The data set that we obtained contained the following attributes of the client profile: credit line, sex, education, marital status, and age. About their credit card payment history, for six months it compiled the amount of payments that the client missed, the amount owed, and the amount payed. This gave a single vector of 23 dimensions. The last column of the data set stated whether the client defaulted on the credit card on the month after the information was compiled.

The data set had about 110,000 rows, each of them with 24 columns. This made it much bigger than the other data sets we have looked at before.

After we pre-processed the data set, by changing the yes/no column that we wanted to predict by 1 and -1, we created a model through a Decision Tree Classifier algorithm, available through the Scikit-learn Python library. We then ran Aequitas of all three versions on this data set and retrained the model with the collected discriminatory inputs. We then proceeded to repeat this process on the resulting models two more times.

C. Results

These are the results of Aequitas on the default of credit card clients model.

When ran Aequitas Random,

When ran Aequitas Semi-Directed,

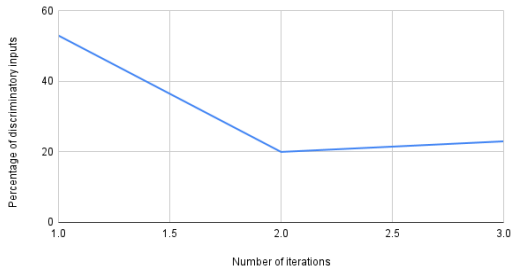

Figure 4. The plot of percent bias found versus the number of iterations.

When ran Aequitas Fully-Directed,

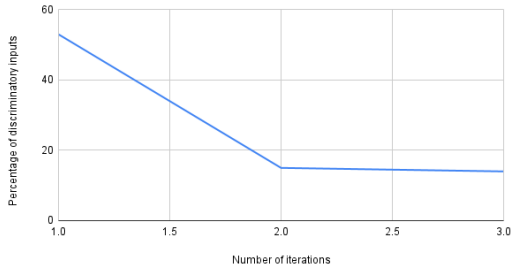

Figure 5. The plot of percent bias found versus the number of iterations.

D. Discussion

As it was mentioned in Section I, we question whether Aequitas Fully Directed and Aequitas Semi-Directed are versions of the algorithm that are worth using in a application setting, as the amount of inputs that they give are not much more than the ones that Aequitas Random gives. Furthermore, in the retraining process, we are not guaranteed to use all of

the inputs. In fact, the core of the algorithm is that we only use a portion of this inputs on each iteration.

We wonder whether it might be more worthwhile to run Random, Semi-Directed, and Fully Directed in succession, based on how many inputs are generated by each, and whether that quantity is sufficient to retrain the model. However, there are also drawbacks to this approach, as it could take much longer to run all three types of Aequitas than it is to just run a single Fully Directed version.

Finally, on our personal machine, each run of Aequitas Fully Directed took about half an hour, which is much longer than we would like it to be for a web application.

REFERENCES

[1]Tejashvi, "Employee Future Prediction," (2021).
[2]D. Dua and C. Graff, "UCI Machine Learning Repository: Banknote Authentication Data Set," (2017).
[3]D. Dua and C. Graff, "UCI Machine Learning Repository: Haberman's Survival Data Set," (2017).
[4]Y. I-Cheng, "default of credit card clients Data Set," (2021).

A Time Complexity Analysis for our Comps Group Implementation of Aequitas Fully Directed.

Michael Worrell, Yemi Shin, Yunping Wang, and Juanito Zhang Yang
Department of Computer Science, Carleton College, 300 North College Street, Northfield, MN, 55057
(Dated: March 16, 2022)

For our Comps this trimester, we did research into and worked to implement a modified version of the machine learning fairness algorithm “Aequitas.” For this analysis write up, we will be discussing the time complexity of our own implementation of the algorithm presented in the original Aequitas paper.

Time complexity analysis paper preface: What to expect, and Why.

The original Aequitas paper did not include a time complexity analysis of their own algorithm. We therefore would like to present our own time complexity analysis of our own implementation.

It is also important that we record the time complexity analysis of our own implementation, because we made fundamental changes to the original algorithm for the sake of runtime optimization that drastically changed the time complexity that would have resulted from perfectly implementing the original Aequitas paper.

When measuring the time complexity, we are measuring it in terms of Big O notation. However, we will be measuring the time complexity of our algorithm in the *average* case. The reason we are doing this is because several of the data structures we are using, such as sets (which in Python are implemented as dictionaries) have a time complexity of $O(1)$ in the average case, but a time complexity of $O(n)$ in the worst case.

To better represent what elements of Aequitas are likely to take up the largest number of computations, we will treat the average case as the standard for data structures, while keeping everything else in terms of the worst case.

This time complexity writeup will only discuss the time complexity of Aequitas Fully Directed. One reason for this is because out of all of the versions of the code we implemented, Aequitas Fully directed is the one we spent the most time optimizing, testing, and fixing throughout our comps project, and as a result is more in tune with the spirit of our Comps project.

Variable definitions used throughout this time complexity analysis

Here are variables that will be used throughout the remainder of this time complexity analysis. For any additional questions, read the

- g = *global_iteration_limit,* as defined in our Aequitas configuration file.
- l = *local_iteration_limit,* as defined in our Aequitas configuration file.
- S = The set of sensitive features
- Si = The i_th sensitive feature
- Ri = The set of values that are possible for the sensitive feature S_i
- $O(model.predict)$ = The time it takes to return classifications for an input in the machine learning model.
- P = The set of parameters in any given input.

Time Complexity Analysis: Helper function & common operations time complexities.

While the *Aequitas_fully_directed_sklearn* algorithm works using custom functions, it also relies upon previously designed functions and operations that have their own previously established time complexities. In this part, we will record their time complexities for reference when calculating the *Aequitas_fully_directed_sklearn* time complexity.

Time Complexity Analysis, Part 1: NumPy Python Library

The proof of concept implementation that we built our version of Aequitas from used NumPy (Numerical Python) functions. The NumPy library is what our implementation uses in the processing of multidimensional array objects, which Aequitas uses to store and manipulate inputs. The NumPy functions used in *Aequitas_fully_directed_sklearn* and their time complexities are as follows:

- np.asarray(input) has a time complexity of $O(|P|)$. This is true because this function is copying the data into a new array object, which runs in $O(n)$ time.
- np.delete(input, index) has a time complexity of $O(|P|)$. This works in an essentially identical way to removing items from an array, which runs in $O(n)$ time.
- np.reshape(input) has a time complexity of $O(1)$. This is because reshape appears to change how the stored data is interpreted, not how the data itself is stored.

Time Complexity Analysis, Part 2: Python Set Operations

Our Aequitas implementation uses sets to help keep track of which inputs were discovered as being discriminatory. The Python set operations used and their time complexities are as follows:

- Adding an item to a set in python has a time complexity of $O(1)$.
- Checking whether an item is present in a set in python has a time complexity of $O(1)$.

Time Complexity Analysis, Part 3: Python List Operations

Our Aequitas implementation uses sets to help keep track of which inputs were discovered as being discriminatory. The Python list operations used in Aequitas_fully_directed_sklearn and their time complexities are as follows:

- Appending an item to a python list has a time complexity of $O(1)$.
- Retrieving an item from a python list has a time complexity of $O(1)$.

Time Complexity Analysis, Part 4: Miscellaneous functions

There are also miscellaneous functions used throughout several important functions. The time complexities of these functions that are used in Aequitas_fully_directed_sklearn are as follows:

- random.randint(0, n) has a time complexity of $O(1)$.
- random.choice(array *or* list) has a time complexity of $O(1)$.
- Fully_Direct.normalize_probability has a time complexity of $O(|P|)$.

Time Complexity Analysis, Part 5: Basinhopping

Basinhopping, as we use it in our implementation, has the following parameters:

- func (function to put under test)
- x0 (the starting inputs for the basinhopping process)
- stepsize
- take_step (algorithm determining how perturbation takes a step, or a constant value)
- minimizer_kwargs (minimizer used)
- niter (number of iterations)

The basinhopping function works by taking some input, deciding how large of a step to take and where to take the step, perturbing the input by that step amount, using the function on the perturbed inputs using the minimizer to receive a score it is attempting to minimize, and then repeating until conditions inherent to the minimizer are met, up to a maximum of *niter* times.

For each call to basinhopping in our algorithm, stepsize is constant. So, the only parameters that we need to consider for runtime purposes are func, x0, take_step, minimizer_kwargs, and niter. From this, we see that the time complexity of the basin hopping algorithm is

$$O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs))).$$

Time Complexity Analysis: Aequitas_fully_directed_sklearn.

Aequitas_fully_directed_sklearn can be split into two parts. The first of these parts performs global discovery to find discriminatory inputs, and the second of these parts performs local discovery to find disriminatory inputs in the vicinity of discrimnatory inputs found during global discovery.

Aequitas_fully_directed_sklearn, Part 1: Global discovery time complexity

For the global discovery portion of the Aequitas algorithm we implemented, Aequitas runs the basinhopping function one time. This means that the global discovery algorithm has a time

complexity of $O(basinhopping)$. The next thing to do is calculate the time complexity of basinhopping during this global discovery process.

Aequitas_fully_directed_sklearn, Part 1, Subsection 1: basinhopping

Here, the basin hopping parameters that are relevant to the time complexity are:

- func ← Fully_Direct.evaluate_global
- take_step ← Fully_Direct.global_discovery
- minimizer_kwargs ← minimizer (more specifically, a L-BFGS-B method minimizer).
- niter ← g

Using the definition from *Time Complexity Analysis, Part 5,* we find that the time complexity for each runthrough of the basin hopping algorithm during local search becomes

$$O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$$

Expanding upon the time complexity equation we found in "*Aequitas_fully_directed_sklearn, Part 2",* the time complexity of global discovery becomes

$$O(g*(O(fully\_direct.evaluate\_global)+O(fully\_direct.global\_discovery)+ O(minimizer)))$$

.

We will now describe the time complexity of each relevant function that is used within basinhopping for *aequitas_fully_directed_sklearn* global discovery, and will then bring them together after each function has been evaluated in detail.

Aequitas_fully_directed_sklearn, Part 1, Subsection 2: Fully_Direct.global_discovery

First, global discovery generates a random seed, which has a time complexity of $O(1)$. Then, the function random.randint(low, high) repeats for each parameter in the provided input, and uses low and high corresponding to the range of possible values for the chosen parameter. Since random.randint(low, high) has a time complexity of $O(1)$, this means that generating a new global input adds $O(|P|)$ to the time complexity of *Fully_Direct.global_discovery*.

Lastly, Setting the sensitive feature value for the new input to 0 has an average case time complexity of $O(1)$. This means that this function has a time complexity of $O(|P|)$.

Aequitas_fully_directed_sklearn, Part 1, Subsection 3: Fully_Direct.evaluate_global

For the first part of this function, the function makes calls to np.asarray, np.reshape, and model.predict are made, as well as adds to a set and checks whether an item is already present in

a set. This has an average time complexity of $O(O(model.predict) * |P|)$. If the original input is present in the global_discovery discriminatory input set, the function ends here.

If the original input is not present in the global_discovery discriminatory input set, the function loops through the $|R_{input}|$ inputs that differ only in the given sensitive feature. For each new input, calls to np.asarray, np.reshape, and model.predict are made. This has an average time complexity of $O(O(model.predict) * |P|)$ for each new input tested. This means that the entirety of this internal loop has a time complexity of$O(|R_{input}| * O(model.predict) * |P|)$.

This continues until a discriminatory input is discovered, or until the function proves that the original input is not discriminatory for the sensitive feature selected. In the event that a discriminatory input is discovered, adding the original input to the global_discovery discriminatry input set and appending it to the global_discovery input list has an average time complexity of $O(1)$.

All of this tells us that our **Fully_Direct.evaluate_global function** has three cases to consider.

- In the first case, the original input is already present in the global_discovery discriminatory input set. This case has a time complexity of

  $O(O(model.predict) * |P|)$.

- In the second case, the original input is not present in the global_discovery discriminatory input set, and is discovered to be discriminatory in regards to the selected sensitive feature. This case has a time complexity of $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|) + O(1))$, which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

- In the third case, the original input is not present in the global_discovery discriminatory input set, and is discovered to *not* be discriminatory in regards to the selected sensitive feature. This case has a time complexity of

  $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|)$,

  which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

Putting all of this together, this means that the function *fully_direct.evaluate_global* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

<u>Aequitas_fully_directed_sklearn, Part 1, Subsection 4: minimizer</u>

The minimizer Aequitas uses is L-BFGS-B. According to Dewi Retno Sari Saputro and Purnami Widyaningsih, that particular minimizer algorithm has a time complexity of $O(mn)$, where *m* is

the number of inputs (in our case, the number of parameters for the input to test), and *n* is the maximum number of variable metric corrections used to define the limited memory matrix. However, the value of *n* is locked to a default value for the scipy.optimize.minimize("L-BFGS-B") function. This means that the time complexity of the minimizer function is $O(|P|)$.

Aequitas_fully_directed_sklearn, Part 1, Subsection 5: Bringing everything together

Now that we know the time complexity for all of the functions and methods utilized in the global discovery stages of our implementation, we can put them together. The new time complexity can be expanded and then simplified as follows:

$\rightarrow O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$

$\rightarrow O(g*(O(Fully\_Direct.evaluate\_global)+O(Fully\_Direct.global\_discovery)+ O(minimizer)))$

$\rightarrow O(g * (O(|R_{input}| * O(model.predict) * |P|) + O(P) + O(|P|)))$

$\rightarrow O(g * O(|R_{input}| * O(model.predict) * |P|))$

$\rightarrow O(g * |R_{input}| * |P| * O(model.predict))$

Aequitas_fully_directed_sklearn, Part 2: Global discovery time complexity

For the local discovery portion of the Aequitas algorithm we implemented, Aequitas runs the basinhopping function once *per discriminatory input found during global discovery*. This means that the local discovery algorithm has a time complexity of $O(g * O(basinhopping))$. The next thing to do is calculate the time complexity of basinhopping during this global discovery process.

Aequitas_fully_directed_sklearn, Part 2, Subsection 1: basinhopping

Here, the basin hopping parameters that are relevant to the time complexity are:

- func ← Fully_Direct.evaluate_local
- take_step ← Fully_Direct.local_perturbation
- minimizer_kwargs ← minimizer (more specifically, a L-BFGS-B method minimizer).
- niter ← l

Using the definition from *Time Complexity Analysis, Part 5,* we find that the time complexity for each runthrough of the basin hopping algorithm during local search becomes

$O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$

Expanding upon the time complexity equation we found in *Aequitas_fully_directed_sklearn, Part 2,* the time complexity of local discovery becomes

$$O(local\_iteration\_limit*(O(fully\_direct.evaluate\_local)+O(fully\_direct.local\_perturbation)+O(minimizer)))$$

.

We will now describe the time complexity of each relevant function that is used within basinhopping for *aequitas_fully_directed_sklearn* local discovery, and will then bring them together after each function has been evaluated in detail.

Aequitas_fully_directed_sklearn, Part 2, Subsection 2: Fully_Direct.evaluate_local

For the first part of this function, the function makes calls to np.asarray, np.reshape, and model.predict are made, as well as adds to a set and checks whether an item is already present in a one of two separate sets, one from the global discovery step, and one from the local discovery step. This has an average time complexity of $O(O(model.predict) * |P|)$. If the original input is present in either the global_discovery discriminatory input set or the local_discovery discriminatory input set, the function ends here.

If the original input is not present in either discriminatory input set, the function loops through the $|R_{input}|$ inputs that differ only in the given sensitive feature. For each new input, calls to np.asarray, np.reshape, and model.predict are made. This has an average time complexity of $O(O(model.predict) * |P|)$ for each new input tested. This means that the entirety of this internal loop has a worst case time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

This continues until a discriminatory input is discovered, or until the function proves that the original input is not discriminatory for the sensitive feature selected. In the event that a discriminatory input is discovered, adding the original input to the local_discovery discriminatry input set and appending it to the local_discovery input list has an average time complexity of $O(1)$.

All of this tells us that our ***Fully_Direct.evaluate_local*** **function** has three cases to consider.

- In the first case, the original input is already present in either of the global_discovery or local_discovery discriminatory input sets. This case has a time complexity of

  $O(O(model.predict) * |P|)$.

- In the second case, the original input is not present in either discriminatory input set, and is discovered to be discriminatory in regards to the selected sensitive feature. This case has a time complexity of $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|) + O(1))$, which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

- In the third case, the original input is not present in the either discriminatory input set, and is discovered to *not* be discriminatory in regards to the selected sensitive feature. This case has a time complexity of

  $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|)$,

  which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

Putting all of this together, this means that the function *Fully_Direct.evaluate_local* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Aequitas_fully_directed_sklearn, Part 2, Subsection 3: Fully_Direct.evaluate_input

This particular function behaves identically to the function *Fully_Direct.evaluate_local*, with a few notable exceptions. When it runs, it does not check or add to any discriminatory input sets. Rather, it returns a different numerical result once the input has been identified as discriminatory or non-discriminatory. This changes the cases to consider as follows:

- The first case no longer exists, so there is no time complexity to consider.
- In the second case, the original input is discovered to be discriminatory in regards to the selected sensitive feature. This case has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.
- In the third case, the original input is discovered to *not* be discriminatory in regards to the selected sensitive feature. This case has a time complexity of

  $O(|R_{input}| * O(model.predict) * |P|)$.

Putting all of this together, this means that the function *Fully_Direct.evaluate_input* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Aequitas_fully_directed_sklearn, Part 2, Subsection 4: Fully_Direct.local_perturbation

First, this function copies input bounds, one per parameter index, to a new array. This part has a time complexity of $O(|P|)$. Then, it performs between 2 and 3 random.choice() operations that each have a time complexity of $O(1)$, which since means the time complexity for *fully_direct.local_perturbation* is still at $O(|P|)$.

After this, *fully_direct.local_perturbation* runs *fully_direct.evaluate_input*, which has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Then, it performs constant time operations until it normalizes the probability for the perturbed input, which has a time complexity of $O(|P|)$.

Bringing this all together, we find that *Fully_Direct.local_perturbation* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Aequitas_fully_directed_sklearn, Part 2, Subsection 5: minimizer

The minimizer function used for local discovery is identical to the minimizer function used for global discovery (see *Aequitas_fully_directed_sklearn, Part 1, Subsection 4)*. This means that the time complexity of the minimizer is $O(|P|)$.

Aequitas_fully_directed_sklearn, Part 2, Subsection 6: Bringing everything together

Now that we know the time complexity for all of the functions and methods utilized in the local discovery stages of our implementation, we can put them together. The new time complexity can be expanded and then simplified as follows:

$\rightarrow O(g * O(basinhopping))$

$\rightarrow O(g * O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs))))$

$\rightarrow O(g*O(l*(O(Fully\_Direct.evaluate\_local)+O(Fully\_Direct.local\_perturbation)+ O(minimizer))))$

$\rightarrow O(g*O(l*(O(|R_{input}|*O(model.predict)*|P|)+O(|R_{input}|*O(model.predict)* |P|) + O(|P|))))$

$\rightarrow O(g * O(l * O(|R_{input}| * O(model.predict) * |P|)))$

$\rightarrow O(g * l * |R_{input}| * |P| * O(model.predict))$

Aequitas_fully_directed_sklearn, Part 3, Subsection 1: Combining both parts.

Now we can look at the time complexities of global discovery and local discovery alongside each other. Combining both to the same equation, the time complexity becomes

$O(g*|R_{input}|*|P|*O(model.predict))+O(g*l*|R_{input}|*|P|*O(model.predict))$,

which can be simplified to a time complexity of $O(g * l * |R_{input}| * |P| * O(model.predict))$.

Aequitas_fully_directed_sklearn, Part 3, Subsection 2: Time complexity with multiple sensitive features

In our implementation, we allow the user to test multiple sensitive features at once. However, we do it slightly differently than the original Aequitas paper suggests we should. We chose to do this in order to improve runtime to make the site easier to use and more accessible.

To do this, we run the algorithm all over again on different sensitive features. This translates into a time complexity of

$$O\left(\sum_{\forall i:S_i \in S} g * l * |P| * O(model.predict) * |R_i|\right).$$

This is great, but it could look cleaner. After pulling all of the constants out of the summation, the time complexity for Aequitas Fully Directed when testing for multiple sensitive features simultaneously in our implementation can be rewritten in a way we believe is more informative.

$$O\left(g * l * |P| * O(model.predict) * \left(\sum_{\forall i:S_i \in S} |R_i|\right)\right)$$

We present to you, the worst case time complexity for our implementation of the Aequitas Algorithm that allows for multiple non-binary sensitive features while making runtime optimizations.

Citations Page

Udeshi, S., Arora, P., & Chattopadhyay, S. (2018). Automated Directed Fairness Testing. ArXiv. https://doi.org/10.48550/ARXIV.1807.00468

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2..

Van Rossum, G. (2020). The Python Library Reference, release 3.8.2. Python Software Foundation.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17(3), 261-272.

Saputro, D. R. S., & Widyaningsih, P. (2017). Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method for the parameter estimation on geographically weighted ordinal logistic regression model (GWOLR). In AIP Conference Proceedings. THE 4TH INTERNATIONAL CONFERENCE ON RESEARCH, IMPLEMENTATION, AND EDUCATION OF MATHEMATICS AND SCIENCE (4TH ICRIEMS): Research and Education for Developing Scientific Attitude in Sciences And Mathematics. Author(s). https://doi.org/10.1063/1.4995124

Hello Comps Group Members,

I am not positive whether what I am experiencing is a side effect of how the code should run, or if something has been improperly implemented, although I believe that it might be the latter.

Regardless of whether using binary of non-binary sensitive attributes, within the double for loops in *evaluate_global* and *evaluate_local*, *i* and *j* do not iterate through a number of values that increass with the number of total values possible for the sensitive parameter. What I mean by this is, for the following code:

```
for i in self.input_bounds[self.sensitive_param_idx]:
    for j in self.input_bounds[self.sensitive_param_idx]:
        if i < j:
            <Rest of evaluate code here>
```

The ***j*** loop will run at most twice for each time the ***i*** loop runs, regardless of the total number of sensitive parameters, and the ***i*** loop will run at most twice for each time **evaluate_global** or **evaluate_local** is called

In what I believe is a side effect of this, as the code stood on Thursday (not sure if these changes have been fixed since), local and global search only add discriminatory inputs where the sensitive parameter equals 0. Here is what I believe might be causing these effects.

The data stored at **self.input_bounds** is in the form of an array with two values each, corresponding to the number of possible values that the parameter could contain.

Using **"Employee.csv"** as an example, **self.input_bounds[self.sensitive_param_idx]** would return **[0, 1]** on the **"Gender"** sensitive attribute, **[0, 2]** on the **"Education"** sensitive attribute, and **[0, 6]** on the **"JoiningYear"** sensitive attribute.

The double nested for loops take this information instead of running **i = 0, i = 1, i = 2 … i = 6** for the sensitive attribute **"JoiningYear"**, it runs **i = 0, i = 6** instead.

An in depth example of this can be observed when running on the sensitive parameter **"Education"**. There are only three possible values, and the result one gets when retrieving **self.input_bounds[self.sensitive_param_idx]** for this parameter is **[0, 2]**.

Instead of the **for** loops iterating through

i = 0 & j = 0	i = 0 & j = 1	i = 0 & j = 2
i = 1 & j = 0	i = 1 & j = 1	i = 1 & j = 2
i = 2 & j = 0	i = 2 & j = 1	i = 2 & j = 2

The **for** loops instead iterate through

i = 0 & j = 0	i = 0 & j = 2
i = 2 & j = 0	i = 2 & j = 2

Because of the way the **if** statement is written, the code following the **if** statement proceeds for
i = 0 & j = 2
Instead of proceeding for the values of

i = 0 & j = 1	i = 0 & j = 2
i = 1 & j = 2	

What I wanted to check was, is this behavior intentional?

If it is, could one of you help me understand why Aequitas needs to work this way, so that I can become more on the right track in understanding how our code should function?

If it is not intentional, could you work to fix this in the main code? If this is the case, it is most likely messing with our code's ability to successfully identify all discriminatory inputs for non-binary sensitive parameters, as well as giving us an optimistic outlook as to the legitimate runtime of our code.

Best, Michael Worrell.

Why/How evaluate_input, evaluate_global, and evaluate_local need/needed to be changed
- Michael Worrell, to my fellow members of Comps Group Aequitas Fairness

Part I: Improper for loop use in our evaluation functions

Here is the pseudocode for the search algorithm used in Aequitas.

Algorithm 2 Local Search

```
1:  procedure LOCAL_EXP(disc_inps, P, P_disc, Δ_v, Δ_pr)
2:      Test ← φ
3:      Let P' = P \ P_disc
4:      Let σ_pr[p] = 1/|P'| for all p ∈ P'
5:      Let σ_v[p] = 0.5 for all p ∈ P'
6:      for I ∈ disc_inps do
7:          ▷ N is the number of trials in local search
8:          for i in (0, N) do
9:              Select p ∈ P' with probability σ_pr[p]
10:             Select δ = −1 with probability σ_v[p]
11:             ▷ Note that I is modified as a side-effect of modifying I_p
12:             I_p ← I_p + δ
13:             ▷ 𝕀^(d) extends I with all values of P_disc
14:             𝕀^(d) ← {I' | ∀p ∈ P \ P_disc. I_p = I'_p}
15:             if (∃I, I' ∈ 𝕀^(d), |f(I) − f(I')| > γ) then
16:                 ▷ Add the perturbed input I
17:                 Test ← Test ∪ {I}
18:             end if
19:             update_prob(I, p, Test, δ, Δ_v, Δ_pr)
20:         end for
21:     end for
22:     return Test
23: end procedure
```

However, this algorithm covers all calls to basinhopping, not just evaluate_local. A singular basinhopping iteration uses this portion of the above algorithm:

```
7:          ▷ N is the number of trials in local search
8:          for i in (0, N) do
9:              Select p ∈ P' with probability σ_pr[p]
10:             Select δ = −1 with probability σ_v[p]
11:             ▷ Note that I is modified as a side-effect of modifying I_p
12:             I_p ← I_p + δ
13:             ▷ 𝕀^(d) extends I with all values of P_disc
14:             𝕀^(d) ← {I' | ∀p ∈ P \ P_disc. I_p = I'_p}
15:             if (∃I, I' ∈ 𝕀^(d), |f(I) − f(I')| > γ) then
16:                 ▷ Add the perturbed input I
17:                 Test ← Test ∪ {I}
18:             end if
19:             update_prob(I, p, Test, δ, Δ_v, Δ_pr)
20:         end for
```

The great thing about basinhopping is that it utilizes the local_perturbation and global_perturbation functions to perform the following lines of the algorithm:

```
9:              Select p ∈ P' with probability σ_pr[p]
10:             Select δ = −1 with probability σ_v[p]
11:             ▷ Note that I is modified as a side-effect of modifying I_p
12:             I_p ← I_p + δ
```

As the pseudocode states, note that *I* is the perturbed input that is being plugged into the evaluate_local function. This means that evaluate_local essentially performs the following steps:

```
14:             𝕀^(d) ← {I' | ∀p ∈ P \ P_disc. I_p = I'_p}
15:             if (∃I, I' ∈ 𝕀^(d), |f(I) − f(I')| > γ) then
16:                 ▷ Add the perturbed input I
17:                 Test ← Test ∪ {I}
18:             end if
19:             update_prob(I, p, Test, δ, Δ_v, Δ_pr)
```

Now, this is where the issue of non-binary sensitive attributes was causing our implementaion to not only perform extra, unneeded work, but actively go against the algorithm presented in the original paper.

What this is saying is that if *I* is the input that reached this step, $\mathbf{I^{(d)}}$ (sorry, couldn't figure out how to type out the double I in google docs) is the set of all inputs that are identical to *I* with the exception of their sensitive parameter, with the number of entries in $\mathbf{I^{(d)}}$ equaling the number of values possible for the sensitive attribute.

What *should* happen in evaluate_local and evaluate_global is that it should check whether *the entry I* and some entry *I'* in $\mathbf{I^{(d)}}$ cause the model to return different values, in which case *I* should be added to the local discrimination set. What we *are* doing is checking whether some entry *I'* in $\mathbf{I^{(d)}}$*, and another* entry *I'* in $\mathbf{I^{(d)}}$, cause the model to return different values, in which case *whichever sensitive parameter of I' has the lowest value* would be added to the local discrimination set..

This was not limited to evaluate_local.. We used our previous method of identifying discriminatory inputs in evaluate_local, evaluate_global, and *evaluate_input* (more on that in part III).

We also used this method of identifying discriminatory input in Retrain_Sklearn, which led to an increased number of inputs being identified as discriminatory.

In my branch, michael_runtimeOptimizations, I have adjusted how each of the evaluate functions (in Aequitas as well as during Retraining_Sklearn)

operates in order to keep our code more in line with the original Aequitas paper.

Part II. Remnants of binary sensitive parameter cases in our code.

The evaluate_input functions within Aequitas_Fully_Directed, Aequitas_Semi_Directed, and Aequitas_Random were hard-coded for a binary column-to-be-predicted that had potential values of -1 and 1 only. Because we are working with a binary column-to-be-predicted that can have values of 0 or 1, the previous code was often inaccurate when evaluating whether a given input was discriminatory.

Part III. Retraining_Sklearn

One last change that needed to be made to our Aequitas code was to include “threshold” in the evaluate_input function within Retrain_Sklearn. Before, similar to the evaluate_input functions mentioned in Part II, Retrain_Sklearn was hard-coded for a binary column-to-be-predicted that had potential values of -1 and 1 only, and worked without the need for threshold. Because we want to provide users the option to change the threshold value, retrain_sklearn and several functions contained therein have an additional parameter for threshold (see .py file for exact parameter location for each function, should the need arise before the code freeze).