FairTear: Automated Probabilistic Analysis on Dataset Models

Yash Patel, Zachary Liu

***Abstract*—Given the extent to which machine learning algorithms have come to characterize lives, both on a daily and long-scale basis, the study of their ingrained biases is much in order. Many tools have emerged to understand such biases, both those that explicitly look at the underlying classifier code (white-box) and those that are agnostic thereof (black-box). White-box tools can provide greater insight, but are typically limited in the types of models they can analyze. A new tool, FairSquare, provides a method of applying white-box techniques to more complex models. However, since FairSquare requires a new classifier syntax and knowledge of an underlying population model, there was much left to be desired as an end-user.**

We present a tool, FairTear, which provides a clean UI through which end users can feed in their classifier and view its analysis result from the FairSquare tool. Our tool automates both the process of generating the population model and the process of converting a classifier to the FairSquare syntax. In turn, the user is fully abstracted from the FairSquare back-end, allowing them to determine the fairness of his algorithm without any additional knowledge than what is contained in their code.

FairTear is capable of making use of nearly all the supported FairSquare functionality, supporting multi-level conditioning of population model features and different feature distributions (Gaussian and multi-step uniform). FairTear also integrates with the popular scikit-learn Python machine learning package, supporting several of its classifiers (decision trees, SVMs, and neural networks) in addition to additional preprocessing steps (StandardScaler). In doing so, we hope to allow a variety of end-users, from academia and industry alike, to take advantage of our system in real-world machine learning pipelines.

Tests revealed full automation on all ends (i.e. supporting each of the classifiers referenced above), with fairness results being displayed on the front-end and an appropriate classifier decomposition visible on the back-end. In line with that, we considered further extensions to both our tool and FairSquare. These largely revolve around supporting a greater extent of the sklearn library, including additional distributions, preprocessing features, and classifiers.

I. Introduction

With the introduction of machine learning algorithms to mainstream applications, the issues of inherent unfairness and biases arises as a significant issue. Machine learning has grown into positions in which they are being used to decide moments in people's lives, ranging from major to seemingly minute, such as deciding from whether they will be given bail to their online shopping experiences [2] [4]. It, therefore, stands to reason that steps should be taken towards the ends of mitigating issues that may arise as a result of biases in such algorithms, beginning with initially detection.

Another critical point is the adoption of such systems. With the recent surge criticisms regarding online algorithms, such as Facebook's "echo-chamber" news feed, people have become more aware and skeptical of their use. ("Understanding the reasons behind predictions is, however, quite important in assessing trust" [7]) Were it possible to automate the detection and analysis of fairness of an algorithm, the process of instating a corresponding legislative department to ensure such fairness would be much more streamlined.

This "detection" of bias, however, is ill-defined. Being of a nascent research field, the ML bias community has separated into studying distinct topics, from understanding theoretical incongruities of fairness metrics to using publicly available data as proxies to sensitive attributes [2] [3]. Despite this branching, one of the key themes through all of studies is the ambiguity of the notion of "fairness." No one metric clearly or universally captures the seemingly natural notion of "fairness." Two of the main distinctions between such metrics is whether they relate to group fairness or individual fairness. Specifically, "individual fairness... dictates that similar inputs must result in similar outputs; and group fairness... dictates that a particular subset of inputs must have a similar aggregate output to the whole" [1]. In line with that, tools have been developed to allow researchers and developers to not only automate the testing of fairness within their algorithm, but further allow them to define the metric of interest in ascribing a fairness classification.

In particular, such tools often focus on one of two main "forms" of explanation: one focusing on overall models, the other on individual decisions. The former seeks to obtain a high-level, macroscopic view of the model, whereas the latter seeks to obtain a low-level, microscopic view. As stated in the de-facto paper on interpretability, these were framed respectively as seeking "transparency, i.e. how does the model work?" and "post-hoc explanations, i.e. what else can the model tell me?" [5] The use cases of each vary widely, with the former largely being the concern of the end developers of the model (i.e. for purposes of debugging) and the latter for end users (i.e. for explaining to them the steps that were followed in reaching the final conclusion).

In line with that, tools have emerged in two main forms: black-box and white-box, which largely align with the distinction made by Lipton in his paper. These are named based on the scope of information available to the algorithm. In particular, the former seeks to understand bias through analysis of algorithms *without* access to the underlying code, whereas the latter *has* access to some underlying structure. One key advantage of black-box tools is that they can be applied without regards to the underlying development framework. For example, a recent research effort developed "LIME, a novel explanation technique that explains the predictions of any

classifier in an interpretable and faithful manner, by learning an interpretable model locally around the prediction...LIME samples instances, gets predictions using f, and weighs them by the proximity to the instance" [7]. In other words, LIME relies on the principal of continuity, where the general behavior of a classifier can be generally determined by finding its behavior locally on specific features. That is to say, were a classifier given an image of a cat, we would expect its output to remain largely unchanged given $\leq 5\%$ of the input pixels were altered.

Given that this black-box technique is universally applicable, insight can be gained about where biases are by highlighting the most pivotal features in making the final decision. This procedure, by which the essential features in making classifications is made explicit, is what has largely come to characterize black-box interpretability models. An example of LIME's use was in an image classification task, where the researchers were able to identify the pieces of image that were most critical in classifying it as particular categories, i.e. "Electric Guitar" or "Labrador." Another tool, similarly black-box in nature, is "Gradient-weighted Class Activation Mapping (Grad-CAM), [which] uses the gradients of any target concept (say logits for 'dog' or even a caption), flowing into the final convolutional layer to produce a coarse localization map highlighting the important regions in the image for predicting the concept" [8].

Unfortunately, such tools have limitations. First, they are developed with a specific notion of "fairness" in mind. The two examples previously discussed primarily accomplished their goals by highlighting parts of the data most important to the classification. In doing so, it becomes straightforward for developers to *qualitatively* analyze the fairness, but making a *quantitative* statement of fairness is difficult to achieve using such means. White-box methods, however, fulfill these two major voids of interest that are not served by black-box tools. They are able to capture different notions of fairness and make quantifiable assessments of fairness. As a result, more white-box tools have begun to emerge. Seeing that the main advantage of black-box methods is their universal applicability, a universal white-box method would prove useful.

One such tool that has recently emerged is FairSquare. This tool aims to allow people to "use it to verify a class of fairness properties for a broad spectrum of decision-making programs generated from real-world datasets" [1]. In fact, while this tool was not the first of its type, it is more capable than those previously developed, due to the underlying method employed for estimating probabilities and for performing numerical integration. Previous tools against which FairSquare was compared, such as PSI, did not perform as well since many terminated without an answer [6]. The main reason behind the lack of termination was its seeking of "exact inference in probabilistic programs with both continuous and discrete random variables" [6]. While this would presumably extend to the optimal point of verification, much of its applications are restricted to toy problems lacking in significant complexity. The approximation technique employed in FairSquare, using a combination of forming precise decision boundaries coupled with SMT solvers, allowed for handling a significantly more complex set of problems, despite sacrificing a fraction of accuracy. For this reason, it was of interest to extend this particular tool over others, though the system developed herein was designed in a modular fashion so as to allow extensions beyond FairSquare.

As for the specific extensions considered herein, we consider the primary objective cited by the FairSquare authors themselves: "We envision a future in which those who employ algorithmic decision-making in sensitive domains are required to prove fairness of their processes. Towards this vision, our goal in this paper is to develop an automated technique that can prove fairness properties of programs" [1]. Though FairSquare is significantly more automated than any of its manual counterparts, it still requires significant manual work to compile and run a program in its custom language. The pipeline of FairSquare can be broken into two distinct steps: analyzing the population models of underlying model variables and combining it with the classifier. In order to be available for analysis by FairSquare, both the model and classifier must be expressed as a special .fr code format with a custom Python-like syntax.

It, therefore, came to our attention that two major substeps within this pipeline could be automated to make the tool easier to use:

- **Curating a population model:** Users often do not have the population model on-hand while developing their ML pipelines. In particular, the framing of FairSquare seems to cater to a modelling approach to ML as opposed to the discriminatory approach that has now taken hold of a majority of the community. In other words, it is no longer the responsibility of the developers or researchers themselves to come up with the relations between input features: this has now been deferred to the algorithms. In a similar manner, it was of interest to push the burden of coming up with population models from the developers to the algorithms.
- **Syntactically framing the models as an .fr file:** Similarly, it is poor UX to necessitate a completely new framework to be used just for one piece of functionality. For this reason, while we rely on the .fr backend, this will be abstracted away from the user, making it so that they can easily use the tool with existing classifiers.

In summary, the primary goal was to develop a better UX to encourage algorithm developers to make better use of FairSquare. One of the authors' primary objectives in developing FairSquare is "a future in which those who employ algorithmic decision-making in sensitive domains are required to prove fairness of their processes" [1]. In line with accomplishing that goal, we envision this tool as having as simple an interface, so as to pose no significant upfront cost (in terms of developing time) for end users. To those ends, we have developed and deployed the **FairTear** tool, which has been open-sourced and is available at: https://github.com/yashpatel5400/fairtear, appropriately named for being able to tear apart input datasets and classifiers to calculate and determine their paired fairness. The paper is organized as follows:

- **Implementation Backend:** We begin by describing the

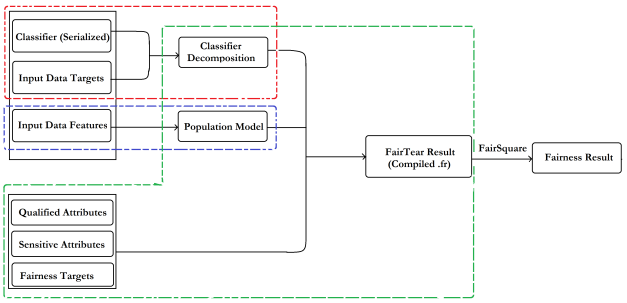

Fig. 1. The overall FairTear pipeline can be broken into three components: (1: Blue) Dataset modelling is the step that takes the features and constructs parameterized models from them; (2: Red) Classifier decomposition takes the serialized classifier and targets to construct a classifier in the standard "decision-tree" like format desired by FairSquare; and (3: Green) Combines the different attributes/targets, population model, and compiled classifier and outputs the .fr file. This is finally fed into FairSquare (not considered in the pipeline), from which we obtain our final result.

details of how the back-end system functions, including how the dataset modelling, classifier decomposition, and overall fairness pipeline steps were automated. We then describe how these were integrated into a custom user interface through which the FairTear system can be used via a web browser.

- **Results:** We go through various outputs that were computed using the FairTear pipeline.
- **Discussion:** Going through the results, we discuss what they demonstrate about the current abilities and limitations of the deployed system.
- **Future Work:** We discuss further extensions that can be implemented on top of the current version to further encourage adoption by the mainstream.

II. Implementation

The implementation details, as elaborated in Figure 1, are separated into separate sections, in which further elaboration is provided for each step. These include mathematical background/explanation coupled with some important implementation details. What is additionally important to note is the modularity of the overall system, where the population model and classifier decomposition do not directly rely on FairSquare for their functionality. This allows FairTear to be easily adapted to work with new analysis tools similar to FairSquare.

A. Dataset Modelling

The overall objective of dataset modelling was to create an end-to-end pipeline taking the dataset in as input and producing the corresponding population model as its output. Looking at an example output provided in FairSquare, it becomes evident that there arise many problems beyond independently modelling each of the features in the dataset.

In the following section, we consider the points outlined in Figure 2. We begin by discussing the simplest version of the population model, and then use this to discuss how following steps build upon this algorithm, in the following order:

```
def popModel():
    sex = step([(0,1,0.3307), (1,2,0.6693)])
    if sex < 1:
        capital_gain = gaussian(568.4105, 24248365.5428)
        if capital_gain < 7298.0000:
            age = gaussian(38.4208, 184.9151)
            education_num = gaussian(10.0827, 6.5096)
            relationship = step([(0,1,0.0491), (1,2,0.1556), (2,3,0.4012), (3,4,0.2589), (4,5,0.0294),
            (5,6,0.1058)])
        else:
            age = gaussian(38.8125, 193.4918)
            education_num = gaussian(10.1041, 6.1522)
            relationship = step([(0,1,0.0416), (1,2,0.1667), (2,3,0.4583), (3,4,0.2292), (4,5,0.0166),
            (5,6,0.0876)])
    else:
        capital_gain = gaussian(1329.3700, 69327473.1006)
        if capital_gain < 5178.0000:
            age = gaussian(38.6361, 187.2435)
            education_num = gaussian(10.0817, 6.4841)
            relationship = step([(0,1,0.0497), (1,2,0.1545), (2,3,0.4021), (3,4,0.2590), (4,5,0.0294),
            (5,6,0.1053)])
```

Fig. 2. Example population model that was analyzed by FairSquare. Points to note are the partitioning of the feature space by conditionals (referred to simply as "partitioning" elsewhere if not ambiguous), ability to have arbitrarily many levels of conditionals, and presence of two distribution types (Gaussian and step).

- Feature space partitioning (single-level)
- Feature space partitioning (multi-level)
- Handling multiple distribution types

1) Single-Level Partitioning: We begin with single-level partitioning of the feature space. What is meant by "single-level" is that there is simply a conditional (i.e. $x < A$) and its complement ($x \geq A$). In the multi-level expansion we later consider, this is extended to arbitrary numbers of partitions, i.e. we can have $[x < A_0, A_0 \geq x < A_1, ..., A_n \geq x]$ for arbitrarily large n. Further, we assume that the distributions for *all* variables are Gaussian, allowing us to take advantage of Gaussian independence.

Having prefaced the partitioning, we now turn to the material at hand. First, we establish notation:

- $D = [x_1, x_2, ..., x_m]$: Input dataset, consisting of individual data points x_i
- $f_1, f_2, ..., f_n$: Features (i.e. income, height, etc...). Of note is that these are the columns of the transpose of D, i.e. $D^T = [f_1, f_2, ...f_n]$
- μ_{f_i}, σ_{f_i} are respectively the mean and standard deviation of f_i

It comes to note that there are now three main problems to resolve:

1) Determine whether to partition one variable on another
2) (If so) Determine what value to use as the partitioning value (threshold)

These two problems are, in fact, quite related. For the first, we refer to a method that is typically employed in decision trees. First, we fix a given feature f_i. The partitioning values we will consider for f_i will be:

$$\theta = [\mu_{f_i} - \sigma_{f_i}, \mu_{f_i} - \sigma_{f_i}/2, \mu_{f_i}, \mu_{f_i} + \sigma_{f_i}/2, \mu_{f_i} + \sigma_{f_i}]$$

Note that these were chosen arbitrarily with the option of being able to extend this range. Fix any one of these paritioning values and denote it θ_k (for threshold). These were chosen to balance the time necessary for computation while maintaining a high level of accuracy. Returning to the matter at hand, by "partitioning values," we mean the thresholds considered for conditioning, i.e. having A by the "partitioning value" on

feature f_i would entail having (in the final .fr file) to have something of the form:

```
if f_i < A:
    ...
else:
    ...
```

We then consider each of the features f_j for $j > i$[1]. For any pair of f_i, f_j, partition f_j into the two sets corresponding to the f_i threshold. That is, split f_j into f_{j_1}, f_{j_2} where f_{j_1} is the entries for which $f_i \leq \theta_k$ and f_{j_2} for $f_i > \theta_k$.

From these sets, we now calculate the entropy and see whether a sufficient information gain was achieved to warrant making a partition. In other words, we see the information gained by making a partition of some f_i on another f_j to determine whether to partition or not. Since these are datapoints coming from a discrete dataset, we calculate entropy using a "binning" method.

As a brief aside, this procedure proceeds as follows. For some set $X = [x_1, x_2, ..., x_n]$, to calculate its entropy, partition the set into k bins (split evenly across the dataset range), for some fixed k[2]. These datasets each contain some $[\ell_1, \ell_2, ...\ell_k]$ elements in them, where $\sum_i \ell_i = n$. From there, we can approximate the probabilities as $[p_1, p_2, ..., p_k]$, where each $p_i = \ell_i/n$. Thus, $\sum_i p_i = 1$. From here, we can employ the standard entropy calculation, which proceeds as $\sum_i p_i \log_2(p_i)$.

Algorithm 1 Binned entropy calculation used to calculate information gain and, therefore, whether or not partitioning should be performed. Note that X refers to a set that is passed in.

procedure BINNEDENTROPY(X)
  $k \leftarrow 25$
  $partitions \leftarrow$ *X partitioned into k bins*
  $sizes \leftarrow$ *sizes of the partitions*
  $probs \leftarrow sizes/N$
return $\sum_i (probs_i \log_2(probs_i))$

To finally determine whether or not the information gain was sufficient, we calculate the pre- and post-partitioned entropies. The former is simply $E_{before} := BinnedEntropy(f_j)$ whereas the latter is a weighted sum of the partitioned sets, namely based on their sizes. Denote $N_1 := |f_{j_1}|$, $N_2 := |f_{j_2}|$ and similarly $E_1 := BinnedEntropy(f_{j_1}), E_2 := BinnedEntropy(f_{j_2})$:

$$E_{after} := \frac{1}{N_1 + N_2}(N_1 \cdot E_1 + N_2 \cdot E_2)$$

Information gain is defined as $IG := E_{after} - E_{before}$. We now consider an arbitrary threshold parameter (γ for Information *Gain*). For our purposes we chose $\gamma = .125$ empirically. If this $IG > \gamma$, we consider this a *valid* partition, i.e. one that will potentially be made. Having determined the information gain for this θ_k partition value, repeat for each of the other θ_k and determine which has the maximum information gain. Make a partition on the threshold that results in the maximum information gain. If, however, no information gain exceeds γ, we will choose to withhold, saying it is instead better to treat the two variables as independent of one another. Thus, the algorithm can be summarized as follows:

Algorithm 2 Partitioning algorithm employed to determine, given two feature sets f_i, f_j, whether a partition is advantageous and, if so, what the optimal choice of a threshold is. Note that we use $\gamma = .125$ below, although this choice is arbitrary. It is also an implementation detail as to return just the threshold vs. the threshold and partitions (we chose latter to avoid repeated work).

procedure PARTITION(f_i, f_j)
  $IG_{best} \leftarrow -\infty$
  $\theta_{best} \leftarrow None$
  $partition_{best} \leftarrow None$
  $\gamma \leftarrow .125$
  $\theta \leftarrow [\mu_{f_i} - \sigma_{f_i}, \mu_{f_i} - \sigma_{f_i}/2, \mu_{f_i}, \mu_{f_i} + \sigma_{f_i}/2, \mu_{f_i} + \sigma_{f_i}]$
  for $k = 1 : 5$ **do**
    $f_{j_1} \leftarrow [], f_{j_2} \leftarrow []$
    for $\ell = 1 : n$ **do**
      if $f_i[\ell] \leq \theta_k$ **then**
        $f_{j_1}.append(f_j[\ell])$
      else
        $f_{j_2}.append(f_j[\ell])$
    $N_1 \leftarrow length(f_{j_1})$
    $N_2 \leftarrow length(f_{j_2})$
    $E_{old} \leftarrow BinnedEntropy(f_j)$
    $E_1 \leftarrow BinnedEntropy(f_{j_1})$
    $E_2 \leftarrow BinnedEntropy(f_{j_2})$
    $E_{new} \leftarrow 1/(N_1 + N_2)\,(N_1 * E_1 + N_2 * E_2)$
    $IG \leftarrow E_{new} - E_{old}$
    if $IG > \gamma$ *and* $IG > BestIG$ **then**
      $IG_{best} \leftarrow IG$
      $\theta_{best} \leftarrow \theta$
      $partition_{best} \leftarrow (f_{j_1}, f_{j_2})$
  return $\theta_{best}, partition_{best}$

Note that there are also further details involved in the algorithm, i.e. disregarding partitions that result in one of the two sets (i.e. f_{j_1} or f_{j_2}) being too small.

2) Multi-Level Partitioning: After this single-level partitioning has been employed, the natural extension is determine the extension to multi-level partitioning, in which we allow for several partitions to employed on the same variable. In fact, this extension is a very natural one from the single-level implementation. In particular, we effectively recursively apply the above partitioning algorithm indefinitely until no information gain is made that exceeds the threshold set (same threshold γ employed above). While we say "indefinitely," for sake of termination, we have a maximum recursion limit

[1]Note that, the reason we do *not* choose to do each of the other j is because we want to avoid repeating work. It follows that, were f_j a good partitioning variable for f_i, the opposite would also follow, meaning this partition would have been established earlier in the algorithm

[2]For our purposes, the implementation had $k = 25$. For a sufficiently large dataset, this seems to work appropriately, but a smaller dataset may have several empty bins, resulting in issues further down the calculation

(which we denote ρ_{max}), such that the algorithm will automatically terminate after this max number of levels has been reached. Once again, empirically this was determined/set to be $\rho_{max} = 5$.

At each step of the partitioning, determine the partition from current set of partitions (of f_j) that has the max entropy (initially this will just be the f_j itself), which we denote as f_k. Take the values of f_i corresponding to this f_k. Namely, f_k will be some subset of f_j that runs through a subset of its indices, $I \subset [1 : n]$. Consider the relevant subset of f_i as those indexed by I, and denote this f_ℓ. That is to say, $f_\ell := f_i[I]$. Now take these two sets, and run the previously ascribed partitioning algorithm on them. Namely, determine the result of $Partition(f_k, f_\ell)$. If it is the case this returns empty ($None$), terminate. Otherwise, replace the entries in the set with the new partition and recur. This is summarized in the following algorithm:

Algorithm 3 Multi-level partitioning algorithm employed to determine, given two feature sets f_i, f_j, the best number of partitions (ranges from $0..5$) and best choices for said number. Note that, for clarity, we assume the standard Partition(f_i, f_j) algorithm to return *both* the threshold *and* partitions.

procedure MULTIPARTITION(f_i, f_j)
  $\rho_{max} \leftarrow 5$
  $partitions \leftarrow [f_j]$
  $\Theta \leftarrow []$
  for $q = 1 : \rho_{max}$ **do**
    $E_{min} \leftarrow -\infty$
    $f_k \leftarrow None$
    for $i = 1 : len(partitions)$ **do**
      $E \leftarrow BinnedEntropy(partitions[i])$
      if $E < E_{min}$ **then**
        $E_{min} = E$
        $f_k \leftarrow partitions[i]$
    $f_\ell \leftarrow f_i$ *entries for* f_k
    $\theta, partition_{best} \leftarrow Partition(f_\ell, f_k)$
    if $partition_{best}$ *is not None* **then**
      insert θ *at* i *in* Θ
      $partitions[i] = partition_{best}$
    else
      Break
  return $\Theta, partitions$

Thus, the multi-level extension is not much more layered on the previous implementation described.

3) Handling Multiple Distributions: Multiple distributions is independent of the implementations deployed above. Namely, we perform the fits after partitioning the data. That is to say, once the data has been separated into different sets, we can individually model each. FairSquare currently supports two types of distributions: Gaussian (normal) and step (extension of uniform). The step distributions are extensions of uniform, in that they are of the form $[(step_1, p_1), ..., (step_k, p_k)]$, where each p_i is the probability of falling in $step_i$. We, of course, need $\sum_i p_i = 1$, from which the distribution is constructed.

For these, we made use of the Kolmogorov-Smirnov (KS) test, which is a metric of goodness of fit, determining the "distance" between two distributions. This only necessitates having the CDF of the distribution at hand. Thus, we considered the best Gaussian fit (with well-known fitting methods) and the best partition steps of data (for the number of partitions ranging from $1 : 5$) for the step fit, found their corresponding KS distances, and took the minimum of the two. In this way, the fits for each of the features were established, completing the population model generation.

```
def F():
    if relationship < 1:
       t = 1
    elif relationship < 2:
        if age < 21.5:
            t = 1
        else:
            if age < 47.5:
                t = 1
            else:
                t = 0
    else:
        t = 1
    fairnessTarget(t < 0.5)
```

Fig. 3. Example classifier that was analyzed by FairSquare. What is critical to observe is the structure of the overall program, where there are typically some hybrid variables defined in the beginning followed subsequently by branching decisions.

B. Classifier Decomposition

In addition to the population, the classifier too had to be automatically decomposed to create an end-to-end pipeline. For this first iteration of the system, we support decision trees, SVMs (support-vector machines), and neural networks, all of which must be from the standard sklearn Python package. While we investigate each of these three classifier decompositions separately more in depth in the following subsections, we wish to briefly establish some points ahead of the fact that relate to all of these, namely with reference to 3. Specifically, the form of the final .fr output (i.e. those that FairSquare natively handles) is of branching conditionals defined over the space of the input features. In other words, the classifier must be decomposed into a level-by-level constraining set ending in the classification, much in the way a decision tree is structured, from which FairSquare can do its decision boundary calculations/analysis.

To this end, the problem of "classifier decomposition" essentially boils down to determining how a classifier can be expressed fully in terms of such hybrid decision boundaries. We expound on this very point for the three cases at hand in the following sections.

1) Decision Trees: Decision tree classifiers learn a chain of nested decision rules to classify each data point. We can represent this in FairSquare using nested branching conditionals. This results in a simple, interpretable program output, from which the structure of the classifier can be seen at a glance. This type of classifier is a natural fit for the branching structure of a FairSquare program. In addition, decision trees are also a powerful classifier that works well for many problems, making it useful to support in our system.

Unfortunately, the program produced from a DecisionTreeClassifier can be arbitrarily long depending on the complexity of the trained classifier, with an asymptotic complexity linear in the size of the input dataset. As a result, if the classifier is too complex, the compiled program may take prohibitively long to evaluate in FairSquare. By default, sklearn builds a tree with only pure nodes, resulting in perfect training accuracy but a highly complex and overfit classifier. To keep the complexity low, it is important to train the decision tree with reasonable terminating conditions for the tree. Choosing appropriate `max_depth`, `min_samples_split`, and/or `min_samples_leaf` options when creating the DecisionTreeClassifier will help produce a shorter program.

We implemented support for compiling a trained DecisionTreeClassifier from the sklearn package. The trained classifier contains an internal Tree structure which represents the decision rules used to classify examples. We recursively navigate through this structure in order to build, line-by-line, the equivalent FairSquare program. This is stored using a tree structure containing compiled code snippets for each node. Each node has two cases:

- *Case 1: split node*
  At a split node, the classifier branches to a child node depending on the value of a particular feature. We compile this into a snippet of the form:

  ```
  if feature > threshold:
      ...
  else:
      ...
  ```

  We then recursively compile each of the children nodes and insert these lines into the conditional bodies.
- *Case 2: leaf node*
  At a leaf node, the classifier makes a decision about the label of the data point. We represent this in the program with the compiled line `label = 0` or `label = 1` based on the decision. No further recursion is performed at this node.

To print out the final compiled program, we return to the root node and recursively output it and its children.

2) SVMs: Support vector machines work by learning a decision boundary between two classes. In our implementation, we support only the LinearSVC class, which finds a hyperplane boundary by learning linear weights for each feature. Instead of using nested conditionals like the decision tree classifier, this classifier relies on computing a weighted sum of the data. This classifier type is very different from the decision tree and is more useful for continuous valued features. It produces a shorter compiled output, with a program size linear in the number of features, since it does not use recursion. This means that the analysis runtime of the final program is more predictable and does not depend on the training parameters.

Converting a LinearSVC classifier to a FairSquare program consists of unpacking the internal weight matrix into individual expressions for each feature. A weighted sum expression is constructed and appended to the output. The intercept parameter is then added to the target variable.

3) Neural Networks: Neural networks have the ability to model arbitrarily complex decision functions, making them powerful classifiers. They are a popular choice for modeling complex systems, but the size and complexity of these networks makes them difficult to analyze and understand by hand. This presents a unique opportunity for FairSquare to make it possible to analytically determine the fairness of a complex neural network.

In our implementation, we support only a restricted class of neural networks that work well with FairSquares program structure. Specifically, our implementation can compile MLPClassifier instances that use the rectified linear unit activation function at each layer. This class supports a perceptron architecture, which uses fully-connected hidden layers. The size of the output program is linear in the number of neurons in the network, including all input, hidden, and output neurons.

We compile the neural network into a FairSquare program by processing each layer at a time, while constructing new variables to represent hidden neurons. Each neuron is compiled in a similar manner to an SVM by unpacking the weight matrix into a linear expression. The relu activation function is then applied to the neuron output by adding the following conditional statements:

```
if value < 0:
    value = 0
```

The final layer of the MLPClassifier uses a logistic activation function to produce the output classification. For training purposes, this is necessary since the logistic function converts the $[-\infty, +\infty]$ range of the pre-activation output to the range $[0, 1]$. However, for our classification purposes, including the logistic output is unnecessary. Since we are only classifying between two classes, our decision boundary is 0.5, meaning that a post-activation value above 0.5 will be classified as label 1 while all other values will be classified as 0. This is equivalent to comparing the pre-activation value with a boundary of 0, since the logistic function is monotically increasing and $\sigma(0) = 0.5$. Thus, we do not compile the logistic function into the program output.

C. Classifier Pipeline

While the decision classifiers we discussed above cover a great deal of simple use cases, there are many additional features that are employed by sklearn users which were deemed crucial to integrate. In particular, a normalization step is typically used before using models such as SVMs or neural networks. This is needed in order to rescale and shift the input data. Our implementation thus supports the StandardScaler preprocessing class, which subtracts the means of features and scales them to unit variance. This is compiled into a program by creating one line for each variable that performs its associated scaling and then saves the result.

In order to include this step as part of a single classifier, we also support the Pipeline class, which provides a way to connect multiple classification steps into a single model. The Pipeline itself is simply an iterable at its core, and we implement it by iterating over its steps in turn, creating new variables to store intermediate steps.

D. User Interface

We created a web-based user interface for FairTear which provides a way for end users to interact with the system and perform fairness analysis. The user can upload their data and classifier files directly through the webapp and specify fairness parameters. They can then run the analysis and view results in their browser.

Our webapp uses a server-client architecture which allows the FairTear service, allowing it to be used via any browser without setup or installation. Most of the heavyweight data processing is performed on the server, while the client performs a small amount of additional processing for the purpose of assisting user input. The app can be easily deployed for use with a one-click deploy button.

Additional documentation regarding the implementation of our user interface, as well as deployment instructions, is available on our GitHub repository (https://github.com/yashpatel5400/fairtear).

E. Server

The FairTear server is implemented in Flask, a Python web server microframework. This framework allowed us to easily integrate our FairTear processing code into the web server. The server exposes a single endpoint, `_analyze_data`, which accepts the following pieces of information from the user:

- data CSV file
- sensitive attribute, conditional, and threshold
- (optional) qualified attribute, conditional, and threshold
- classifier pickle file
- fairness target attribute, conditional, and threshold

Given this information, the server then compiles both a population model and a classifier in the FairSquare language by calling the compilation code described in prior sections. These compiled functions are assembled together into a complete FairSquare program file. Finally, this program file is then handed off to the FairSquare library, which performs the fairness analysis and prints out the fairness result.

All of the output during this process, including compilation statuses and fairness analysis data, is streamed to the users browser as it is ready.

F. Client

The client interface is implemented using standard web technologies (HTML, JS, CSS) along with the Bootstrap, jQuery, and Redux libraries. It provides an easy-to-use interface for end users to interact with the FairTear system and perform analyses without requiring knowledge of the FairSquare language.

When a user opens the webapp, they are presented with a set of input fields. The first section of the page is used to configure the population modeling stage. The user first selects their population dataset by uploading a .csv file. After a file is chosen, the client uses the PapaParse library to load the .csv file in order to read the attributes in the dataset. It then prompts the user to select one of these as the sensitive attribute. The user can then configure the threshold and comparison operator. Optionally, the user can also specify a qualified attribute, or they can choose to omit this parameter.

The second section of the page prompts the user for their classifier model. The user can upload their saved pickle file containing the pretrained model under analysis. They can then specify a desired target attribute name and fairness threshold.

Once the user has entered this information, they can then press the Run Analysis button at the bottom of the page to perform the fairness analysis. This uploads their data to the server, which then proceeds to compile the program and run it through FairSquare on the backend. Analysis progress is streamed to the browser and is displayed in a box on the screen. Once analysis is complete, the final fairness result is shown.

III. Results

As in the FairSquare paper, we consider the performance on the Adult Census Dataset (available at: http://archive.ics.uci.edu/ml/datasets/Adult). The main distinction between the tests/results we wished to obtain through our tests and those conducted in the FairSquare paper is that we treat age as the sensitive attribute rather than sex. To this end, we present the results below and discuss their qualitative interpretations and explanations in the discussions below.

We generated three principal categories of experiments, each of which were performed in multiple stages. These tests are performed on a SVM, NN, and Decision Tree classifier, either on on subsets of the dataet or on the entire dataset. We present the results below, consisting of both the outputs of our program and the final fairness outcome, and discuss their significance in the section to follow. Note that there were some trials that did not terminate in the FairSquare pipeline, which we indicate appropriately in the table.

Each of the decision classifiers were tested across some subset of the input features. That is to say, we consider the first n of the input features (columns of the data) and train on those to predict the income status. In this case, the income is binary, with 0, 1 respectively indicating whether the income does or does not exceed $50,000. Therefore, in any reference to which "subset" the classifier was trained/predicted on in the results or discussion below, assume that this unambiguously refers to a subset of the features.

For sake of completeness, we also include figures that show the entire generated models for the population and each of the three models, although none of these terminated in a reasonable time frame.

Finally, we deployed the FairTear user interface to a publicly-accessible server for testing and demonstration. The webapp is accessible at the following URL: https://fairtear-demo.herokuapp.com/.

We tested the webapp on one of the datasets and SVM classifiers used in our experiments. A screenshot of the resulting output is shown in figure 4. This screenshot shows the three sections of the interface: population modeling, classifier specification, and analysis output. The output section shows the final results of an analysis performed on this dataset and classifier.

TABLE I. RUN RESULTS: CHECKMARK INDICATES COMPLETED RUN, CROSS INDICATES NON-TERMINATION

Features Included	SVM	NN	Decision Tree
3	✓FAIR (5 sec)	✓FAIR (32 sec)	✓UNFAIR (2 sec)
4	✓FAIR (16 sec)	✓FAIR (15 min)	✓UNFAIR (4 sec)
5	✓UNFAIR (∼25 min)	✗	✓UNFAIR (53 sec)

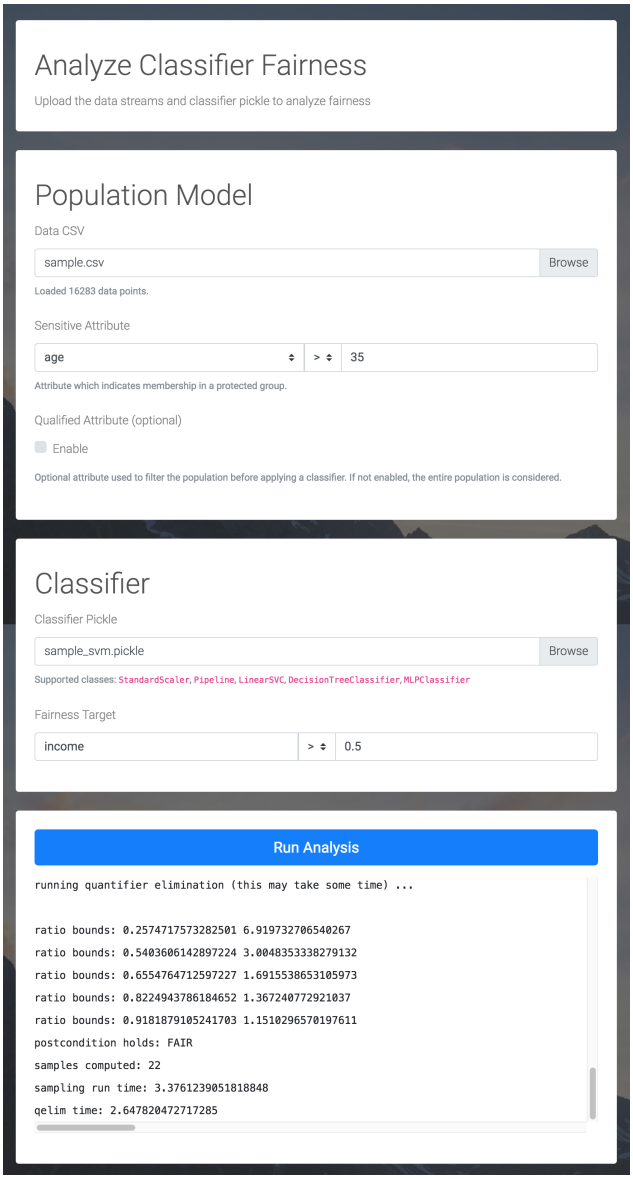

Fig. 4. Screenshot of FairTear webapp after running an analysis.

```
running quantifier elimination (this may take some time) ...

ratio bounds: 0.1605428522678477 1.029109725729709
ratio bounds: 0.4005583329164003 0.9393593398953417
ratio bounds: 0.5241286458073576 0.9308550794176581
ratio bounds: 0.6027876219007527 0.9256235237007806
ratio bounds: 0.6419131370260681 0.9228520631922054
ratio bounds: 0.7487165027436096 0.9214850500834709
ratio bounds: 0.7963365430180189 0.9201149298348474
ratio bounds: 0.796833345026116 0.9190078412448492
ratio bounds: 0.8016788608841782 0.9170677272578213
ratio bounds: 0.8145458981038356 0.9159945072680898
ratio bounds: 0.8156911282696194 0.9132434735115662
ratio bounds: 0.8164723385224314 0.9126795306614389
ratio bounds: 0.8167833744353058 0.9073662425316436
ratio bounds: 0.8197336078244154 0.9068212834035448
ratio bounds: 0.8256178671392277 0.9048025714217226
ratio bounds: 0.834110040490437 0.904206059144771
ratio bounds: 0.8347498092723155 0.904071738253791
ratio bounds: 0.8377991395731299 0.9039523354819797
ratio bounds: 0.8386672640845162 0.903839554669776
ratio bounds: 0.8393243082184534 0.9036375237532883
ratio bounds: 0.8407566146847092 0.9035023171734715
ratio bounds: 0.8424983139597748 0.9034404989685001
ratio bounds: 0.851008139303633 0.9034161614271149
ratio bounds: 0.8511450298201718 0.9033993838284626
ratio bounds: 0.8512981553551859 0.9033747341692334
ratio bounds: 0.851308342242281 0.9033497273512909
ratio bounds: 0.8513121917738657 0.903342903785838
ratio bounds: 0.8513134791810245 0.9000841042317392
ratio bounds: 0.8513147648503152 0.8999612550817294
postcondition does not hold: UNFAIR
samples computed: 66
sampling run time: 19.60383629798889
qelim time: 16.196338891983032
```

Fig. 5. After running any one of the trials, this is the typical output that is produced on the FairSquare back-end. We expect no end user to have to interface with such values (i.e. the ratio bound calculations), given that the primary reason FairTear was developed was to provide an abstraction layer from this complexity. However, debugging fairness in its current state would follow along the lines of attempting to make sense of these bounds and understanding their convergence.

IV. DISCUSSION

We break down the discussion of the previously presented results into four sections, one for the population model and one for each of the classifier types.

A. Population Model

There were two main novel features introduced in this paper with regards to the population model that we wished to study in these results. These were the automated detection of partitioning variables with optimal partition values, as well as the identification of the optimal distribution function for fitting to the underlying features data. We subsequently discuss the progression of the population model as more features were added for fitting classifiers.

Turning our attention first to the input feature partitioning, we observed that the chosen relations align with expectations.

```
def popModel():
    education = gaussian(10.29821, 3.870205)
    if education <= 1.2745:
        education_num = gaussian(6.557401, 0.496695)
    elif education <= 4.3175:
        education_num = gaussian(5.138116, 2.682959)
    elif education <= 7.4365:
        education_num = gaussian(8.063763, 3.792138)
    elif education <= 8.1516:
        education_num = step([(10.5, 11.5, 1.0)])
    elif education <= 10.298:
        education_num = gaussian(13.214806, 0.773483)
    elif 10.298 < education:
        education_num = gaussian(9.941019, 1.675283)
    capital_loss = gaussian(87.30383, 402.954031)
    capital_gain = gaussian(1077.648844, 7385.178677)
    fnlwgt = step([(12285.0, 306769.0, 0.873561), (306769.0, 601253.0, 0.122079), (601253.0, 895737.0, 0.003747), (895737.0,
    1190221.0, 0.000461), (1190221.0, 1484705.0, 0.000152)])
    age = step([(17.0, 31.6, 0.351955), (31.6, 46.2, 0.37502), (46.2, 60.8, 0.201407), (60.8, 75.4, 0.064218), (75.4, 90.0, 0.0074)])
    if age <= 31.762:
        marital_status = step([(0.0, 6.0, 1.0)])
        if marital_status <= 0.85401:
            relationship = gaussian(2.309485, 1.350686)
        elif marital_status <= 2.1632:
            relationship = gaussian(0.902804, 1.845908)
        elif 2.1632 < marital_status:
            relationship = gaussian(2.299121, 1.040819)
    elif 31.762 < age:
        marital_status = gaussian(2.217194, 1.519219)
        if marital_status <= 2.2172:
            relationship = gaussian(0.903378, 1.634768)
        elif 2.2172 < marital_status:
            relationship = gaussian(2.000406, 1.297649)
    race = gaussian(3.665858, 0.848793)
    workclass = step([(0.0, 1.6, 0.08587), (1.6, 3.2, 0.064495), (3.2, 4.8, 0.697031), (4.8, 6.4, 0.112313), (6.4, 8.0, 0.040291)])
    if workclass <= 3.1409:
        occupation = gaussian(4.386643, 4.750282)
    elif 3.1409 < workclass:
        occupation = step([(1.0, 5.333334, 0.422484), (5.333334, 9.666667, 0.234665), (9.666667, 14.0, 0.342851)])
    sex = gaussian(0.669206, 0.4705)
    hours_per_week = gaussian(40.437456, 12.34724)
    native_country = gaussian(36.718867, 7.823662)
    sensitiveAttribute(age > 35)
```

Fig. 6. The overall output of the model when allowed to work over the entire dataset, i.e. *all* 14 of the input features. Much of the behavior we see in this population model, such as the step-wise workclass fit and the partitioning of marital status on age, seems in accord with what would be conventionally deemed as intuitive connections.

```
def popModel():
    fnlwgt = step([(12285.0, 306769.0, 0.873561), (306769.0, 601253.0, 0.122079), (601253.0, 895737.0,
    0.003747), (895737.0, 1190221.0, 0.000461), (1190221.0, 1484705.0, 0.000152)])
    workclass = step([(0.0, 1.6, 0.08587), (1.6, 3.2, 0.064495), (3.2, 4.8, 0.697031), (4.8, 6.4, 0.112313),
    (6.4, 8.0, 0.040291)])
    age = step([(17.0, 31.6, 0.351955), (31.6, 46.2, 0.37502), (46.2, 60.8, 0.201407), (60.8, 75.4, 0.064218),
    (75.4, 90.0, 0.0074)])
    sensitiveAttribute(age > 35)
```

```
def popModel():
    fnlwgt = step([(12285.0, 306769.0, 0.873561), (306769.0, 601253.0, 0.122079), (601253.0, 895737.0,
    0.003747), (895737.0, 1190221.0, 0.000461), (1190221.0, 1484705.0, 0.000152)])
    age = step([(17.0, 31.6, 0.351955), (31.6, 46.2, 0.37502), (46.2, 60.8, 0.201407), (60.8, 75.4, 0.064218),
    (75.4, 90.0, 0.0074)])
    workclass = step([(0.0, 1.6, 0.08587), (1.6, 3.2, 0.064495), (3.2, 4.8, 0.697031), (4.8, 6.4, 0.112313),
    (6.4, 8.0, 0.040291)])
    education = gaussian(10.29821, 3.870205)
    sensitiveAttribute(age > 35)
```

```
def popModel():
    education = gaussian(10.29821, 3.870205)
    if education <= 1.2745:
        education_num = gaussian(6.557401, 0.496695)
    elif education <= 4.3175:
        education_num = gaussian(5.138116, 2.682959)
    elif education <= 7.4365:
        education_num = gaussian(8.063763, 3.792138)
    elif education <= 8.1516:
        education_num = step([(10.5, 11.5, 1.0)])
    elif education <= 10.298:
        education_num = gaussian(13.214806, 0.773483)
    elif 10.298 < education:
        education_num = gaussian(9.941019, 1.675283)
    age = step([(17.0, 31.6, 0.351955), (31.6, 46.2, 0.37502), (46.2, 60.8, 0.201407), (60.8, 75.4, 0.064218),
    (75.4, 90.0, 0.0074)])
    fnlwgt = step([(12285.0, 306769.0, 0.873561), (306769.0, 601253.0, 0.122079), (601253.0, 895737.0,
    0.003747), (895737.0, 1190221.0, 0.000461), (1190221.0, 1484705.0, 0.000152)])
    workclass = step([(0.0, 1.6, 0.08587), (1.6, 3.2, 0.064495), (3.2, 4.8, 0.697031), (4.8, 6.4, 0.112313),
    (6.4, 8.0, 0.040291)])
    sensitiveAttribute(age > 35)
```

Fig. 7. From top to bottom, the population models for the subset of three, four, and five features. These population models are precisely those that were used in *each* of the classifier tests, i.e. there is no direct relation between changing the classifier and the associated population model if the dataset is identical.

Specifically, given that education number is closely related to the level of education a person has attained, the conditioning on these two was likely completely evident. For less direct relations, however, such as age and marital status, we also see that the partitioning was well formed, i.e. those of young age had a greater range in their marital status, whereas almost everyone was married into the middle years of their lives.

Switching to the latter point of novelty, the intuitive understanding behind step distributions is that they should correspond to variables whose underlying distribution is either segmented completely (i.e. a binary variable) or uniformly distributed. In line with that, occupations and workclasses, which occur in finite steps, had their distributions generated as

```
def F():
    scaled_age = (age - 38.5816) * 13.6402
    scaled_workclass = (workclass - 3.8689) * 1.4559
    scaled_fnlwgt = (fnlwgt - 189778.3665) * 105548.3569
    scaled_education = (education - 10.2982) * 3.8702
    scaled_education_num = (education_num - 10.0807) * 2.5727
    if scaled_education_num > 0.9404:
        if scaled_age > -0.6658:
            if scaled_education_num > 1.7178:
                income = 1
            else:
                income = 0
        else:
            if scaled_age > -0.8124:
                income = 0
            else:
                income = 1
    else:
        if scaled_age > -0.3725:
            if scaled_education_num > -0.6144:
                income = 0
            else:
                income = 1
        else:
            if scaled_age > -0.8857:
                income = 0
            else:
                income = 1
    fairnessTarget(income > 0.5)
```

```
scaled_hours_per_week = (hours_per_week - 40.4375) * 12.3472
scaled_native_country = (native_country - 36.7189) * 7.8237
if scaled_relationship > -0.5890:
    if scaled_capital_gain > 0.8119:
        if scaled_age > -1.3256:
            if scaled_fnlwgt > -1.5601:
                if scaled_education_num > -2.7523:
                    if scaled_education_num > 0.1630:
                        income = 1
                    else:
                        if scaled_fnlwgt > -1.4818:
                            if scaled_hours_per_week > -0.3999:
                                if scaled_hours_per_week > 1.4224:
                                    if scaled_workclass > 1.1203:
                                        income = 0
                                    else:
                                        income = 1
                                else:
                                    if scaled_capital_gain > 0.8785:
                                        income = 1
                                    else:
                                        if scaled_capital_gain > 0.8610:
                                            income = 0
                                        else:
                                            income = 1
                            else:
                                if scaled_hours_per_week > -0.4809:
                                    if scaled_capital_gain > 1.5230:
                                        income = 1
```

Fig. 8. The decision tree trained on five of the features (top) and the entire feature space (bottom). Note that the bottom could not be fully included, since its far longer than the space allotted here. In addition, including further parts of the tree would not provide any greater information than that which is captured in this subset.

```
def F():
    scaled_age = (age - 38.5816) * 13.6402
    scaled_workclass = (workclass - 3.8689) * 1.4559
    scaled_fnlwgt = (fnlwgt - 189778.3665) * 105548.3569
    scaled_education = (education - 10.2982) * 3.8702
    scaled_education_num = (education_num - 10.0807) * 2.5727
    income = scaled_age * 0.2060 + scaled_workclass * 0.0324 + scaled_fnlwgt * 0.0178 + scaled_education *
    -0.0102 + scaled_education_num * 0.3270
    income = income + -0.5397
    fairnessTarget(income > 0.5)
```

```
def F():
    scaled_age = (age - 38.5816) * 13.6402
    scaled_workclass = (workclass - 3.8689) * 1.4559
    scaled_fnlwgt = (fnlwgt - 189778.3665) * 105548.3569
    scaled_education = (education - 10.2982) * 3.8702
    scaled_education_num = (education_num - 10.0807) * 2.5727
    scaled_marital_status = (marital_status - 2.6118) * 1.5062
    scaled_occupation = (occupation - 6.5727) * 4.2288
    scaled_relationship = (relationship - 1.4464) * 1.6067
    scaled_race = (race - 3.6659) * 0.8488
    scaled_sex = (sex - 0.6692) * 0.4705
    scaled_capital_gain = (capital_gain - 1077.6488) * 7385.1787
    scaled_capital_loss = (capital_loss - 87.3038) * 402.9540
    scaled_hours_per_week = (hours_per_week - 40.4375) * 12.3472
    scaled_native_country = (native_country - 36.7189) * 7.8237
    income = scaled_age * 0.1524 + scaled_workclass * -0.0101 + scaled_fnlwgt * 0.0179 + scaled_education *
    0.0071 + scaled_education_num * 0.2831 + scaled_marital_status * -0.0967 + scaled_occupation * 0.0148 +
    scaled_relationship * -0.0489 + scaled_race * 0.0284 + scaled_sex * 0.1309 + scaled_capital_gain * 0.7836
    + scaled_capital_loss * 0.1035 + scaled_hours_per_week * 0.1197 + scaled_native_country * 0.0062
    income = income + -0.5243
    fairnessTarget(income > 0.5)
```

Fig. 9. The SVM trained on five of the features (top) and the entire feature space (bottom). Unlike the decision tree or neural network, both outputs are shown in their entirety here.

```
def F():
    scaled_age = (age - 38.5816) * 13.6402
    scaled_workclass = (workclass - 3.8689) * 1.4559
    scaled_fnlwgt = (fnlwgt - 189778.3665) * 105548.3569
    scaled_education = (education - 10.2982) * 3.8702
    hidden_0_0 = scaled_age * -0.5225 + scaled_workclass * -0.0560 + scaled_fnlwgt * -0.0036 +
    scaled_education * -0.8628
    hidden_0_0 = hidden_0_0 + 0.8684
    if hidden_0_0 < 0:
        hidden_0_0 = 0
    hidden_0_1 = scaled_age * -0.8970 + scaled_workclass * -0.0182 + scaled_fnlwgt * -0.0141 +
    scaled_education * 1.1585
    hidden_0_1 = hidden_0_1 + 0.1234
    if hidden_0_1 < 0:
        hidden_0_1 = 0
    hidden_1_0 = hidden_0_0 * 0.8370 + hidden_0_1 * 0.8558
    hidden_1_0 = hidden_1_0 + -1.0518
    if hidden_1_0 < 0:
        hidden_1_0 = 0
    hidden_1_1 = hidden_0_0 * -1.3540 + hidden_0_1 * 0.7336
    hidden_1_1 = hidden_1_1 + -1.1511
    if hidden_1_1 < 0:
        hidden_1_1 = 0
    income = hidden_1_0 * 2.1369 + hidden_1_1 * 1.3391
    income = income + -0.4738
    fairnessTarget(income > 0.5)
```

```
scaled_capital_loss = (capital_loss - 87.3038) * 402.9540
scaled_hours_per_week = (hours_per_week - 40.4375) * 12.3472
scaled_native_country = (native_country - 36.7189) * 7.8237
hidden_0_0 = scaled_age * -0.3470 + scaled_workclass * -0.0139 + scaled_fnlwgt * -0.0911 +
scaled_education * -0.0500 + scaled_education_num * -0.0306 + scaled_marital_status * 0.6707 +
scaled_occupation * 0.1690 + scaled_relationship * 0.0659 + scaled_race * -0.0761 + scaled_sex * -0.1259
+ scaled_capital_gain * 0.1474 + scaled_capital_loss * -0.0250 + scaled_hours_per_week * 0.5775 +
scaled_native_country * -0.0328
hidden_0_0 = hidden_0_0 + 0.5611
if hidden_0_0 < 0:
    hidden_0_0 = 0
hidden_0_1 = scaled_age * 0.0130 + scaled_workclass * 0.0832 + scaled_fnlwgt * 0.1887 + scaled_education
* 0.3024 + scaled_education_num * 0.1085 + scaled_marital_status * -0.1496 + scaled_occupation * -0.1052
+ scaled_relationship * 0.2139 + scaled_race * 0.1120 + scaled_sex * -0.0117 + scaled_capital_gain *
0.3063 + scaled_capital_loss * 0.4731 + scaled_hours_per_week * 0.0436 + scaled_native_country * -0.0386
hidden_0_1 = hidden_0_1 + -0.0578
if hidden_0_1 < 0:
    hidden_0_1 = 0
hidden_0_2 = scaled_age * 0.0125 + scaled_workclass * 0.0195 + scaled_fnlwgt * -0.0011 + scaled_education
* -0.0525 + scaled_education_num * -0.0494 + scaled_marital_status * 0.1481 + scaled_occupation * 0.0208
+ scaled_relationship * 0.4392 + scaled_race * -0.0629 + scaled_sex * -0.0266 + scaled_capital_gain *
2.0750 + scaled_capital_loss * 0.3104 + scaled_hours_per_week * -0.0541 + scaled_native_country * -0.0019
hidden_0_2 = hidden_0_2 + -0.2484
if hidden_0_2 < 0:
    hidden_0_2 = 0
hidden_0_3 = scaled_age * -0.6227 + scaled_workclass * 0.5871 + scaled_fnlwgt * 0.0913 + scaled_education
* 0.1370 + scaled_education_num * -0.3373 + scaled_marital_status * -0.5326 + scaled_occupation * -0.3130
+ scaled_relationship * 0.2230 + scaled_race * 0.0528 + scaled_sex * 0.6869 + scaled_capital_gain *
0.4511 + scaled_capital_loss * -0.0257 + scaled_hours_per_week * -0.2384 + scaled_native_country * -0.0036
hidden_0_3 = hidden_0_3 + 0.6611
if hidden_0_3 < 0:
```

Fig. 10. The neural network (MLP) trained on four of the features (top) and the entire feature space (bottom). The reason we choose to depict the four rather than five case is because five did not terminate, slightly complicated further discussion on the matter. Once again, the bottom result is curtailed for lack of space.

such. Likewise, the alternate case was similarly well-behaved, where any variable we would expect to be continuously distributed was given a Gaussian distribution, including education, capital (gain and loss), and hours per week. Despite this overall reasonable performance, there are some quirks with regards to the classification of certain features.

For example, marital status, education, native country, and one branch of occupation were classified as Gaussian, despite the underlying variable being categorical. Upon further investigation, it became clear that this was largely due to the significant number of possible values in each of those features, making the continuous fit just as appropriate or better than the discrete counterparts.

For example, the "native country" feature had the following set of possible values (each mapped to a numerical label): United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad & Tobago, Peru, Hong, Holand-Netherlands.

This shows that the presence of discrete variables leads to ambiguity in determining what type of underlying distribution is appropriate. An additional option may be implemented going forward to indicate whether a particular feature is discrete or continuous.

Switching over to discussing the progressive compiled population models (i.e. stepping from three features to five features in Figure 7), we notice two main facets with regards to the ideas previously discussed. The first is that independent variables do not change in their distribution, meaning those that were not further conditioned had the identical distributions between the population model of three and four features, as expected. More interestingly, however, the variables were all independent (from a partitioning point of view) until the fifth feature was added in. This seems quite natural in the first case, given that fnlwgt (a feature which represents a weighted sum of demographic features), workclass, and age, are not directly related to one another. While the independence between education and workclass may seem somewhat unintuitive in the second case, this follows because the workclass is quite varied across the same degree and vice versa. That is to say, a given industry will have people from a variety of educational backgrounds, largely owing to the fact that there are people in *many* positions that must be accounted for. In addition, the occupations in the dataset themselves are industries that would likely be of varied backgrounds, consisting of values such as Tech-support, Craft-repair, Other-service, and Sales.

At last, with the fifth feature, we see a level of partitioning, arising from the natural relation between education and education number (the aggregate numerical score to indicate education level). Thus, with further and further extension of the feature space, such partitions become clearly more elaborate.

B. Decision Trees

Seeing that a decision trees are often regarded as one of the most interpretable models in the modern era of discriminatory modelling, it followed quite directly that these could be directly converted into the desired .fr format. As previously described, the classifier output must be of the form of repeated (and often nested) conditionals. Given that decision trees are built with this very structure at their heart, the decomposition was quite natural. We see that the decision tree output consists primarily of nesting conditionals on *independent* variables. Thus, the main point of distinction is that there are no composite variables, as is the case in nearly all the other classifiers. This is to say, given input features A, B, the decision tree creates adjusted versions of those variables $\hat{A}, \hat{B}$ but does not add an additional variable of the form $C = A + B$. This makes interpretation and qualitative understanding of the fairness decision usually more straightforward than is the case in other classifiers.

We now turn our attention to what we can glean from the results of the runs and compiled code. The former makes clear that the decision tree was *significantly* faster than the other classifiers, likely due to the clear decision boundaries that were formed internally by the FairSquare estimator. That is to say, since FairSquare is heavily centered around approximated volumes under regions defined by decision boundaries, the alignment of such boundaries with the input features (owing to

the previously observed fact that no composite features were defined by the decision tree in these instances) likely allowed the computation to follow significantly more smoothly than in the other instances. To this end, it may be worthwhile investigating a manner of adjusting the internal estimation means of FairSquare to take advantage of this alignment and adjust it to the cases of SVMs or NNs.

To ensure the pipeline had appropriate behavior, we attempted to reason through the fairness classification. The unfairness of this decision tree is likely the result of the crucial partitioning of the space by the scaled age in the second level, in turn causing the result to largely depend on this value. Seeking to have the education level be sensitive would likely result in a result of UNFAIR for this same reason.

There are some minor points to note in practical use cases of FairTear with decision trees. The first is that, for many of the decision tree classifiers initially tested, the results were inconclusive and in fact terminated in an unsatisfiability error from FairSquare. After further investigation, it was found that the source of this error was the lack of the different outcomes on different branches, i.e. there were cases in which some branches would both end in $income = 0$. This was fixed with additional parameter tweaking, including specifying a min_samples_split threshold in the DecisionTreeClassifier. Further, the decision trees had significantly more complexity than the other classifiers, producing compiled output that were roughly 25,000 lines of .fr code. Seeing that even 100 lines pushes the boundaries of time feasibility, this seems rather unattainable in FairSquare's current state. Given that fact, we simply circumvented the issue by restricting the maximum depth of the decision tree to 3.

C. SVM

Unlike the decision tree, the SVM is not often touted as universally understandable. While the structure of the overall program itself is relatively reasonable, the final line in which the scaled variables are all used to define a single income variable is quite unreasonable to intuitively understand. However, the compiled code does follow the structure we would generally expect an SVM to have, in which the features are combined into a single, high-dimensional classifier variable, from which the decision boundary is then drawn. The compiled classifier also highlights how the different variables were all measured in significantly different ranges, i.e. each scaled parameter had vastly different values. Thus, an overall takeaway from this particular case is that, while the .fr file itself is typically quite cryptic, there are cases where some semblance of usefulness can be gleaned.

Turning our attention to the compiled results table, we see that the SVM typically had a reasonable time to completion. The cases of two and three input features finished in times not much slower than the decision tree. As soon as five features were introduced, the runtime increased dramatically. This is almost certainly the result of the additionally complexity (noted in the population model discussion) from the fifth feature, namely when partitioning was first introduced. While the decision tree did not have any significant jump, the SVM did, since the high-dimensional final income variable defined lies on an odd intersection of the different feature spaces. Thus, conditional jumps on each of the variables is quite difficult to extrapolate to this final variable, making any complex SVM (i.e. for partitioned spaces) a time drain.

D. Neural Network

We now turn our attention to the neural networks (multilayer perceptrons). In line with the SVM properties discussed above, NN are notorious for being difficult to understand. As expected, the compiled code in Figure 10 is largely cryptic. It is difficult to manually evaluate fairness given these pieces of code, as they combine both composite variables and conditionals. The former (introduction of composite variables) parallels the same issue from SVMs, except the final high-dimensional composite variable is replaced by a number of lower-dimensional hidden layer composites. Further, the presence of conditionals for the hidden layers' thresholding makes decision boundaries irregular, further complicating the work of FairSquare.

As a result of this significant added complexity, we note two main takeaways. First, we notice that the runtime is significantly greater than either the decision tree or SVM classifiers, once again arising from the significantly increased complexity of the decision boundaries. In fact, the lack of convergence of one of the trials seems to suggest that future work should emphasize speeding up this pipeline going forward. Further, in the case of neural networks or similarly complicated architectures, it is difficult to manually corroborate or explain the results, as it is quite difficult to heuristically justify why a certain classifier was deemed fair and another unfair. To that end, we must perform such classifications on simpler, more understandable classifiers, build confidence in the overall system, and extend this trust to the use cases that cannot be fully understood.

V. Future Work

Given that the eventual goal is to automate the implementation of a fairness checker into standard ML development pipelines, we see some extensions to *both* FairSquare and FairTear that will help move towards that end.

A. FairSquare Extensions

While FairSquare is quite elaborate a tool, there are some additional features that can be layered on without critically altering the back-end, largely with application-level extensions. The first is of supporting more probability distributions. While many distributions can be modelled using Gaussian or Stepwise Uniform (i.e. $Step([...])$), including normal, uniform, and binomial, there are many other continuous distributions at hand that cannot, such as exponential and Poisson. Of course, these are largely niche, but should a full automation be desired, all distributions must be covered if we wish to avoid having FairTear be a leaky abstraction. This, additionally, would have to be supported by the support from the FairTear front-end, where we would simply extend the ideas discussed in II-A1,

in which we additionally compute KS distances associated with these additionally supported distributions.

In line with that, additional debugging functionality may be quite critical going forward. As with the advent of new programming languages and frameworks, such as Rust over C/C++, the computer science community as a whole has taken a step towards functional programming. Such frameworks have the key advantage that, once a program has been successfully compiled, it often satisfies many desirable properties, such as thread-safety or concurrency guarantees. An additional facet that makes such languages ideal for development is specificity of debugging, where compile errors can often be significantly more telling than cryptic runtime errors, such as "Segmentation Fault." Thus, in a similar vein, FairSquare could additionally expose results from its operation, whereby the exact source of unfairness arises, whether that be from the classifier or population model itself. The end result, therefore, would point developers in the correct direction for debugging any fairness errors, rather than the current state, where they simply see fair or unfair.

An additional, and perhaps most important, upgrade that can be made is of speed. Specifically, even with the relatively simple cases considered in the results sections above, the results took significant lengths of time to complete, running in excess of half an hour in some cases. Seeing that the cases generated by FairTear were often quite reasonable to see in practical applications, this significant time barrier would impede adoption into large-scale systems. Most of the computation time was spent in a phase where FairSquare is "running quantifier elimination" (read on stdout). This largely corresponds to the estimation process by which the fairness is eventually determined. If either an alternative manner of reaching an estimate were found or an effective manner of distributing this computation, seeing that the ideal setup of this tool would be on some server which the end-user does not have to internally deal with themselves, the experience of using FairTear would be greatly improved.

While there are many natively unsupported mathematical operations in FairSquare, many are typically irrelevant in all but niche machine learning applications, i.e. most trigonometric functions. However, the logistic function is quite crucial in the ending layers of most simple MLP (multi-layer perceptrons), making them quite a useful feature to support. In particular, the key missing component is the support of exponential evalution, i.e. floating point numbers of the form $a * *b$. With this simple extension, a greater range of neural networks could be supported by the overall pipeline.

B. FairTear Extensions

As hinted at above, many of the extensions of FairTear relate to those implemented by FairSquare. That is to say, with additional features integrated in the FairSquare pipeline, FairTear too must be extended to account for those changed. In addition, however, there are some pieces of functionality already supported by FairSquare not yet integrated, due to the diminished quality in results. Specifically, the use of variable conditionals was not yet integrated in FairTear. This refers to when variables are conditioned on one another, i.e. something of the form:

```
A = gaussian(x0,y0)
B = gaussian(x1,y1)
if A < B:
    ...
else:
    ...
```

Namely, where the threshold itself was a variable. While this additional functionality was not much more difficult to implement than the others done herein, its introduction into the pipeline resulted in some suboptimal choices of thresholds and overall program structure (i.e. too many levels of conditionals). We, therefore, chose to air on the side of simplicity, since it both makes the underlying .fr code more understandable and decreases the runtime of FairSquare.

In line with decreasing runtime, the overall FairTear pipeline is quite long, especially for running complex classifiers of modern-day scale. To this end, should we ever desire mainstream adoption, the time barrier must be significantly reduced from its current state, much in the way that compilers, while somewhat significant in large projects, take relatively minimal time to complete their tasks. While there is no clear way to segment fairness calculations from the outset, there may be an underlying way in which the task of calculating fairness could be segmented. In doing so, we could effectively create the equivalent of a "Makefile" for fairness, in which only the parts of the population model or classifier that were changed would require a repeated calculation of fairness.

To follow the same principles of extension discussed in the previous section with FairSquare, there are many ways in which FairTear can be extended. The most direct way is supporting a greater extent of the sklearn package. One specific point of extension is the preprocessors library, namely where additional features other than the StandardScaler can be integrated into the data pipeline that would be standard for end users to have implemented.

In addition, while a great deal of students and elementary studies can be conducted with use of sklearn, many modern applications extend into using packages such as Tensorflow or Torch. In line with the growing popularity of deep learning and its associated frameworks, all tools developed for testing fairness or the like *must* be compatible with these. Although parsing will be more complicated than the MLP parsing that was performed for supporting neural networks herein, and may involve delving into the TensorFlow source code, we believe this to be a crucial next step in the evolution of the FairSquare/FairTear toolkit.

We hope that the open sourcing of our code will make such extensions accessible for developers and researchers who wish to build upon FairTear.

REFERENCES

[1] Albarghouthi, Aws, et al. “FairSquare: Probabilistic Verification of Program Fairness.” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, Dec. 2017, pp. 130., doi:10.1145/3133904.

[2] Chouldechova, Alexandra. “Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments.” *Big Data*, vol. 5, no. 2, 2017, pp. 153163., doi:10.1089/big.2016.0047.

[3] Corbett-Davies, Sam, et al. “Algorithmic Decision Making and the Cost of Fairness.” *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD*, 2017, doi:10.1145/3097983.3098095.

[4] Hannak, Aniko, et al. “Measuring Price Discrimination and Steering on E-Commerce Web Sites.” *Proceedings of the 2014 Conference on Internet Measurement Conference - IMC*, 2014, doi:10.1145/2663716.2663744.

[5] Lipton, Zachary C. ”The Mythos of Model Interpretability.” ArXiV Preprint arXiv:1606.03490 (2016).

[6] Gehr, Timon, Sasa Misailovic, and Martin Vechev. “PSI: Exact Symbolic Inference for Probabilistic Programs.” *International Conference on Computer Aided Verification.* Springer International Publishing, 2016.

[7] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. ”Why Should I Trust You?: Explaining the Predictions of any Classifier.” *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 2016.

[8] Selvaraju, Ramprasaath R., et al. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization.” See https://arxiv.org/abs/1610.02391 v3 (2016).