AI Fairness 36 : An extensible toolkit for detecting and mitigating algorithmic bias

R K E Bellamy
K Dey
M Hind
S C Hoffman
S Houde
K Kannan
P Lohia
J Martino
S Mehta
A Mojsilović
S Nagar
K Natesan Ramamurthy
J Richards
D Saha
P Sattigeri
M Singh
K R Varshney
Y Zhang

Fairness is an increasingly important concern as machine learning models are used to support decision making in high-stakes applications such as mortgage lending, hiring, and prison sentencing. This article introduces a new open-source Python toolkit for algorithmic fairness, AI Fairness 360 AIF360), released under an Apache v2.0 license https://github.com/ibm/aif360). The main objectives of this toolkit are to help facilitate the transition of fairness research algorithms for use in an industrial setting and to provide a common framework for fairness researchers to share and evaluate algorithms. The package includes a comprehensive set of fairness metrics for datasets and models, explanations for these metrics, and algorithms to mitigate bias in datasets and models. It also includes an interactive Web experience that provides a gentle introduction to the concepts and capabilities for line-of-business users, researchers, and developers to extend the toolkit with their new algorithms and improvements and to use it for performance benchmarking. A built-in testing infrastructure maintains code quality.

1 Introduction

Recent years have seen an outpouring of research on fairness and bias in machine learning (ML) models. This is not surprising, as fairness is a complex and multifaceted concept that depends on context and culture. Narayanan described at least 21 mathematical definitions of fairness from the literature [1]. These are not just theoretical differences in how to measure fairness; different definitions produce entirely different outcomes. For example, ProPublica and Northpointe had a public debate on an important social justice issue (recidivism prediction) that was fundamentally about what is the right fairness metric [2–4]. Furthermore, researchers have shown that it is impossible to satisfy all definitions of fairness at the same time [5]. Thus, although fairness research is a very active field, clarity on which bias metrics and bias mitigation strategies are best has yet to be achieved [6].

In addition to the multitude of fairness definitions, different bias-handling algorithms address different parts of the model life cycle; understanding each research contribution, how, when, and why to use it is challenging even for experts in algorithmic fairness. As a result, the general public, fairness scientific community, and artificial intelligence (AI) practitioners need clarity on how to proceed. Currently, the burden is on ML and AI developers, as they need to deal with questions such as "Should the data be debiased?", "Should we create new classifiers that learn unbiased models?" and "Is it better to correct predictions from the model?"

To address these issues, we have created AI Fairness 360 (AIF360),[1] an extensible open-source toolkit for detecting, understanding, and mitigating algorithmic biases. The goals of AIF360 are: to promote a deeper understanding of fairness metrics and mitigation techniques; to enable an open common platform for fairness researchers and industry practitioners to share and benchmark their algorithms; and to help facilitate the transition of fairness research algorithms to use in an industrial setting.

AIF360 will make it easier for developers and practitioners to understand bias metrics and mitigation and to foster further contributions and information sharing. To

Digital Object Identifier: 10.1147/JRD.2019.2942287

[1]https://aif360.mybluemix.net

0018 8646/19   2019 IBM

help increase the likelihood that AIF360 will develop into a flourishing open-source community, we have designed the system to be extensible, adopted software engineering best practices to maintain code quality, and invested significantly in documentation, demos, and other artifacts.

The initial AIF360 Python package implements techniques from eight published papers from the broader algorithm fairness community. This includes over 71 bias detection metrics, 9 bias mitigation algorithms, and a unique extensible metric explanation facility to help consumers of the system understand the meaning of bias detection results. These techniques can all be called in a standard way, similar to scikit-learn's fit/transform/predict paradigm. In addition, there are several realistic tutorial examples and notebooks showing salient features for industry use that can be quickly adapted by practitioners.

AIF360 is the first system to bring the following together in one open-source toolkit: bias metrics, bias mitigation algorithms, bias metric explanations, and industrial usability. By integrating these aspects, AIF360 can enable stronger collaboration between AI fairness researchers and practitioners, helping us to translate our collective research results to practicing data scientists, data engineers, and developers deploying solutions in a variety of industries.

The contributions of this article are as follows:

- extensible architecture that incorporates dataset representations and algorithms for bias detection, bias mitigation, and bias metric explainability;
- design of an interactive Web experience to introduce users to bias detection and mitigation techniques.

This article is organized as follows. In Section 2, we introduce the basic terminology of bias detection and mitigation. In Section 3, we review prior art and other open-source libraries and contributions in this area. The overall architecture of the toolkit is outlined in Section 4, while Sections 5–8 present details of the underlying dataset, metrics, explainer, and algorithms base classes and abstractions, respectively. In Section 9, we review our testing protocols and test suite for maintaining quality code. In Section 10, we describe the design of the front-end interactive experience, and the design of the back-end service. Concluding remarks and next steps are provided in Section 11.

2 Terminology

In this section, we briefly define specialized terminology from the field of fairness in machine learning. A *favorable label* is a label whose value corresponds to an outcome that provides an advantage to the recipient. Examples are receiving a loan, being hired for a job, and not being arrested. A *protected attribute* is an attribute that partitions a population into groups that have parity in terms of benefit received. Examples include race, gender, caste, and religion. Protected attributes are not universal, but are application-specific. A *privileged* value of a protected attribute indicates a group that has historically been at a systematic advantage. *Group fairness* is the goal of groups defined by protected attributes receiving similar treatment or outcomes. *Individual fairness* is the goal of similar individuals receiving similar treatment or outcomes. *Bias* is a systematic error. In the context of fairness, we are concerned with unwanted bias that places privileged groups at a systematic advantage and unprivileged groups at a systematic disadvantage. A *fairness metric* is a quantification of unwanted bias in training data or models. A *bias mitigation algorithm* is a procedure for reducing unwanted bias in training data or models.

3 Related work

Several open-source libraries have been developed in recent years to provide various levels of functionality in learning fair AI models. Many of these deal only with bias detection and provide no techniques for mitigating such bias. Fairness measures [7], for example, provide several fairness metrics, including difference of means, disparate impact, and odds ratio. A set of datasets is also provided, though some datasets are not in the public domain and need explicit permission from the owners to access/use the data. Similarly, FairML [8] provides an auditing tool for predictive models by quantifying the relative effects of various inputs on a model's predictions. This, in turn, can be used to assess the model's fairness. FairTest [9], on the other hand, approaches the task of detecting biases in a dataset by checking for associations between predicted labels and protected attributes. The methodology also provides a way to identify regions of the input space where an algorithm might incur unusually high errors. This toolkit also includes a rich catalog of datasets. Aequitas [10] is another auditing toolkit for data scientists and policy makers; it has a Python library as well as an associated website where data can be uploaded for bias analysis. It offers several fairness metrics, including demographic or statistical parity and disparate impact, along with a "fairness tree" to help users identify the correct metric to use for their particular situation. Aequitas's license does not allow commercial use. Finally, Themis [11] is an open-source bias toolbox that automatically generates test suites to measure discrimination in decisions made by a predictive system.

A handful of toolkits address both bias detection and bias mitigation. Themis-ML [12] is one such repository that provides a few fairness metrics, such as mean difference, and some bias mitigation algorithms, such as relabeling [13], additive counterfactually fair estimator [14], and reject option classification [15]. The repository contains a subset of the methods described in this article. Fairness Comparison [6] is one of the more extensive libraries. It includes several bias

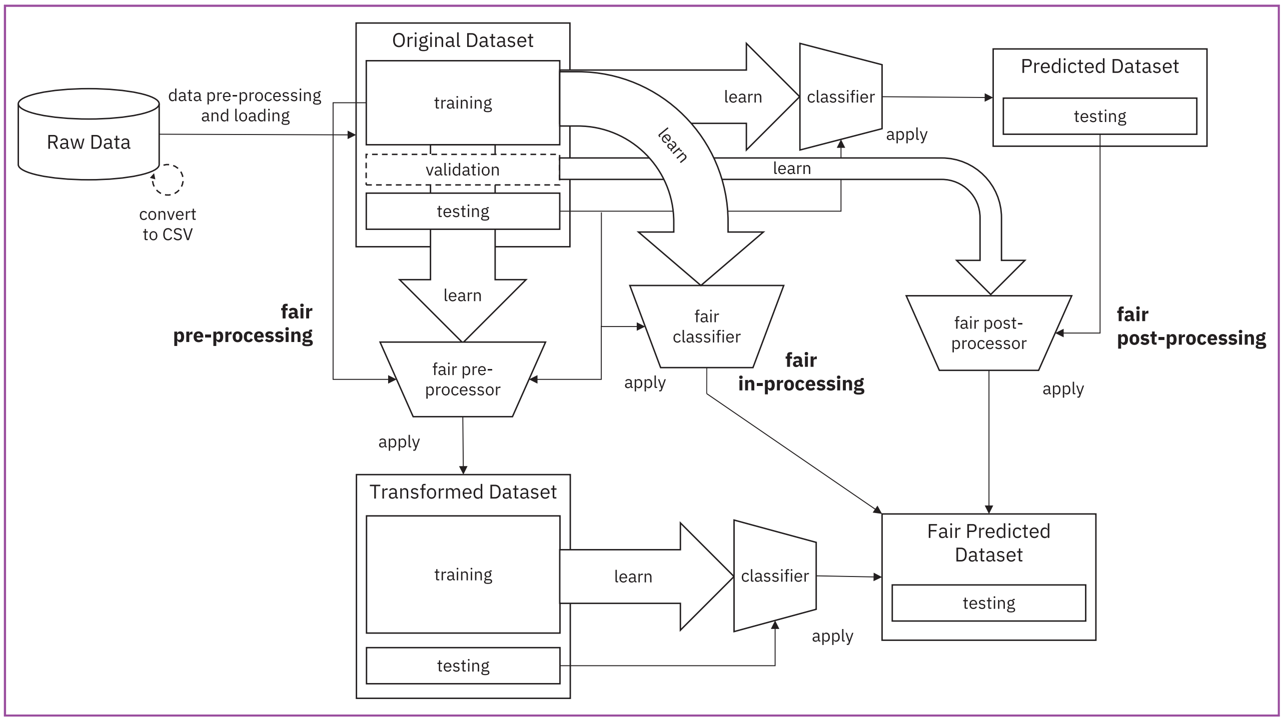

Figure 1

Fairness pipeline. An example instantiation of this generic pipeline consists of loading data into a dataset object, transforming it into a fairer dataset using a fair pre-processing algorithm, learning a classifier from this transformed dataset, and obtaining predictions from this classifier. Metrics can be calculated on the original, transformed, and predicted datasets as well as between the transformed and predicted datasets. Many other instantiations are also possible.

detection metrics and bias mitigation methods, including disparate impact remover [16], prejudice remover [17], and two Naive Bayes [18]. Written primarily as a test-bed to allow different bias metrics and algorithms to be compared in a consistent way, it also allows additional algorithms and datasets.

Our work on AIF360 aims to unify these efforts and bring together in one open-source toolkit a comprehensive set of bias metrics, bias mitigation algorithms, bias metric explanations, and industrial usability. Another contribution of this article is a rigorous architectural design focused on extensibility, usability, explainability, and ease of benchmarking that goes beyond the existing work. We outline these design aspects in more detail in the following sections.

4 Overarching paradigm and architecture

AIF360 is designed as an end-to-end workflow with two goals: 1) ease of use and 2) extensibility. Users should be able to go from raw data to a fair model as easily as possible while comprehending the intermediate results. Researchers should be able to contribute new functionality with minimal effort.

Figure shows our generic pipeline for bias mitigation. Every output in this process (rectangles in the figure) is a new dataset that shares, at least, the same protected attributes as other datasets in the pipeline. Every transition is a transformation that may modify the features or labels or both between its input and output. Trapezoids represent learned models that can be used to make predictions on test data. There are also various stages in the pipeline where we can assess if bias is present using fairness metrics (not pictured) and obtain relevant explanations for the same (not pictured). These will each be discussed as follows.

To ensure ease of use, we created simple abstractions for datasets, metrics, explainers, and algorithms. Metric classes compute fairness and accuracy metrics using one or two datasets, explainer classes provide explanations for the metrics, and algorithm classes implement bias mitigation algorithms. Each of these abstractions is discussed in detail in the subsequent sections. The term “dataset” refers to the dataset object created using our abstraction, as opposed to a CSV data file or a Pandas `DataFrame`.

The base classes for the abstractions are general enough to be useful, but specific enough to prevent errors. For example, there is no functional difference between predictions and ground-truth data or training and testing data. Each dataset object contains both features and labels, so it can be both an output of one transformation and an input to another. More specialized subclasses benefit from inheritance while also providing some basic error-checking, such as determining what metrics are available for certain types of datasets. Finally, we are able to generate high-quality informative documentation automatically by using Sphinx[2] to parse the docstring blocks in the code.

There are three main paths to the goal of making fair predictions (bottom right of Figure 1)—these are labeled in bold: fair pre-processing, fair in-processing, and fair post-processing. Each corresponds to a category of bias mitigation algorithms we have implemented in AIF360. Functionally, however, all three classes of algorithms act on an input dataset and produce an output dataset. This paradigm and the terminology we use for method names are familiar to the machine learning/data science community and similar to those used in other libraries such as scikit-learn.[3] Block arrows marked "learn" in Figure 1 correspond to the `fit` method for a particular algorithm or class of algorithms. Sequences of arrows marked "apply" correspond to `transform` or `predict` methods. Predictions, by convention, result in an output that differs from the input by labels and not features or protected attributes. Transformations result in an output that may differ in any of those attributes.

Although pre-, in-, and post-processing algorithms are all treated the same in our design, there are important considerations that the user must make when choosing which to use. For example, post-processing algorithms are easy to apply to existing classifiers without retraining. By making the distinction clear, which many libraries listed in Section 3 do not do, we hope to make the process transparent and easy to understand.

As for extensibility, while we cannot generalize from only one user, we were very encouraged about how easy it is to use the toolkit when, within days of the toolkit being made available, a researcher from the AI fairness field submitted a pull request asking to add his group's bias mitigation algorithm. In a subsequent interview, this contributor informed us that contributing to the toolkit did not take much time as "*it was very well structured and very easy to follow*."

A simplified UML class diagram of the code is provided in Appendix A for reference. Code snippets for an instantiation of the pipeline based on our AIF360 implementation are provided in Appendix B.

5 Dataset class

The `Dataset` class and its subclasses are key abstractions that handle all forms of data. Training data is used to learn classifiers. Testing data is used to make predictions and compare metrics. Besides these standard aspects of a machine learning pipeline, fairness applications also require associating protected attributes with each instance or record in the data. To maintain a common format, independent of which algorithm or metric is being applied, we chose to structure the `Dataset` class so that all of these relevant attributes—features, labels, protected attributes, and their respective identifiers (names describing each)—are present and accessible from each instance of the class. Subclasses add further attributes that differentiate the dataset and dictate with which algorithms and metrics it is able to interact.

Structured data is the primary form of dataset studied in the fairness literature and is represented by the `StructuredDataset` class. Further distinction is made for a `BinaryLabelDataset`—a structured dataset that has a single label per instance that can only take one of two values: favorable or unfavorable. Unstructured data, which is receiving more attention from the fairness community, can be accommodated in our architecture by constructing a parallel class to the `StructuredDataset` class, without affecting existing classes or algorithms that are not applicable to unstructured data.

The classes provide a common structure for the rest of the pipeline to use. However, since raw data comes in many forms, it is not possible to automatically load an arbitrary raw dataset without input from the user about the data format. The toolkit includes a `StandardDataset` class that standardizes the process of loading a dataset from CSV format and populating the necessary class attributes. Raw data must often be "cleaned" before being used, and categorical features must be encoded as numerical entities. Furthermore, with the same raw data, different experiments are often run using subsets of features or protected attributes. The `StandardDataset` class handles these common tasks by providing a simple interface for the user to specify the columns of a Pandas `DataFrame` that correspond to features, labels, protected attributes, and optionally instance weights; the values of protected attributes that correspond to privileged status; the values of labels that correspond to favorable status; the features that need to be converted from categorical to numerical; and the subset of features to keep for the subsequent analysis. It also allows for arbitrary, user-defined data pre-processing, such as deriving new features from existing ones or filtering invalid instances.

We extend the `StandardDataset` class with examples of commonly used datasets that can be used to load datasets in different manners without modifying code by simply passing different arguments to the constructor. This is in contrast with other tools that make it more difficult to configure the loading procedure at runtime. We currently provide an interface to seven popular datasets: *Adult Census Income* [19], *German Credit* [20], *ProPublica Recidivism COMPAS)* [21], *Bank Marketing* [22], and three versions of *Medical Expenditure Panel Surveys* [23, 24].

Besides serving as a structure that holds data to be used by bias mitigation algorithms or metrics, the `Dataset` class provides many important utility functions and

[2]http://www.sphinx-doc.org/en/master/
[3]http://scikit-learn.org

capabilities. Using Python's == operator, we are able to compare equality of two datasets and even compare a subset of fields using a custom context manager `temporarily_ignore`.[4] The `split` method allows for easy partitioning into training, testing, and possibly validation sets. We are also able to easily convert to Pandas `DataFrame` format for visualization, debugging, and compatibility with externally implemented code. Finally, we track basic metadata associated with each dataset instance. Primary among these is a simple form of provenance tracking: After each modification by an algorithm, a new object is created, and a pointer to the previous state is kept in the metadata, along with details of the algorithm applied. This way, we maintain trust and transparency in the pipeline.

6 Metric class

The `Metric` class and its subclasses compute various individual and group fairness metrics to check for bias in datasets and models.[5] The `DatasetMetric` class and its subclass `BinaryLabelDatasetMetric` examine a single dataset as input (`StructuredDataset` and `BinaryLabelDataset`, respectively) and are typically applied in the left half of Figure 1 to either the original dataset or the transformed dataset. The metrics therein are the group fairness measures of disparate impact and statistical parity difference—the ratio and difference, respectively, of the base rate conditioned on the protected attribute—and the individual fairness measure *consistency* defined by [25].

In contrast, the `SampleDistortionMetric` and `ClassificationMetric` classes examine two datasets as input. For classification metrics, the first input is the original or transformed dataset containing true labels, and the second input is the predicted dataset or fair predicted dataset, respectively, containing predicted labels. This metric class implements accuracy and fairness metrics on models. The sample distortion class contains distance computations between the same individual point in the original and transformed datasets for different distances. Such metrics are used, e.g., by [26], to quantify individual fairness.

A large collection of group fairness and accuracy metrics in the classification metric class consists of functions of the confusion matrix of the true labels and predicted labels, e.g., false negative rate difference and false discovery rate ratio. Two metrics important for later reference are average odds difference (the mean of the false positive rate difference and the true positive rate difference) and equal opportunity difference (the true positive rate difference). Classification metrics also include disparate impact and statistical parity difference, but based on predicted labels. Since the main computation of confusion matrices is common for a large set of metrics, we utilize memoization and caching of computations for performance on large-scale datasets. This class also contains metrics based on the generalized entropy index, which is able to quantify individual and group fairness with a single number [27].

There is a need for a large number and variety of fairness metrics in the toolkit because there is no one best metric relevant for all contexts. It must be chosen carefully, based on subject matter expertise and worldview [28]. The comprehensiveness of the toolkit allows a user to not be hamstrung in making the most appropriate choice. Section 10 of the extended version of this article [29] contains an empirical evaluation that demonstrates how AIF360 can be used for comparisons of bias metrics and mitigation algorithms.

7 Explainer class

The `Explainer` class is intended to be associated with the `Metric` class and provide further insights about computed fairness metrics. Different subclasses of varying complexity that extend the `Explainer` class can be created to output explanations that are meaningful to different user personas. To the best of our knowledge, this is the first fairness toolkit that stresses the need for explanations. The explainer capability implemented in the first release of AIF360 is basic reporting through "pretty print" and JSON outputs. Future releases may include methodologies such as fine-grained localization of bias (described in Section 7.2), actionable recourse analysis [30], and counterfactual fairness [31].

7.1 Reporting

`TextExplainer`, a subclass of `Explainer`, returns a plain text string with a metric value. For example, the explanation for the accuracy metric is simply the text string *"Classification accuracy on count⟩ instances: accuracy⟩",* where count⟩ represents the number of records, and accuracy⟩ the accuracy. This can be invoked for both the privileged and unprivileged instances by passing arguments.

`JSONExplainer` extends `TextExplainer` and produces three output attributes in JSON format:
1) meta-attributes about the metric such as its name, a natural language description of its definition, and its ideal value in a bias-free world; 2) statistical attributes that include the raw and derived numbers; and 3) the plain text explanation passed unchanged from the superclass `TextExplainer`. Outputs from this class are consumed by the Web application described in Section 10.

7.2 Fine grained localization

A more insightful explanation for fairness metrics is the localization of the source of bias at a fine granularity in

[4]A sample snippet of this functionality is as follows:

```
with transf.temporarily_ignore('labels'):
    return transf == pred
```

This returns `True` if the two datasets `transf` and `pred` differ only in labels.

[5]In the interest of space, we omit mathematical notation and definitions here. They may be found elsewhere, including in the documentation for AIF360.

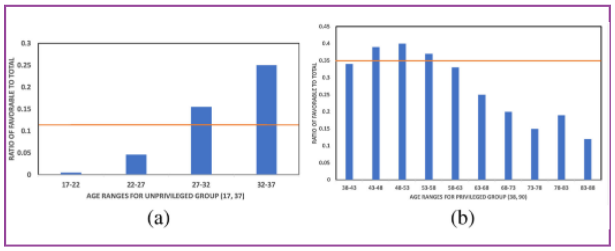

Figure 2

Protected attribute bias localization in (a) younger (unprivileged) and (b) older (privileged) groups in the *German Credit* dataset. The 17–27 range in the younger group and the 43–58 range in the older group would be localized by the approach.

the protected attribute and feature spaces. In the protected attribute space, the approach finds the values in which the given fairness metric is diminished (unprivileged group) or enhanced (privileged group) compared to the entire group. In the feature space, the approach computes the given fairness metric across all feature values and localizes on ones that are most objectionable. **Figure 2** illustrates protected attribute bias localization on the *German Credit* dataset, with age as the protected attribute. **Figure 3** illustrates feature bias localization on the *Stanford Open Policing* dataset [32] for Connecticut, with county name as the feature and race as the protected attribute.

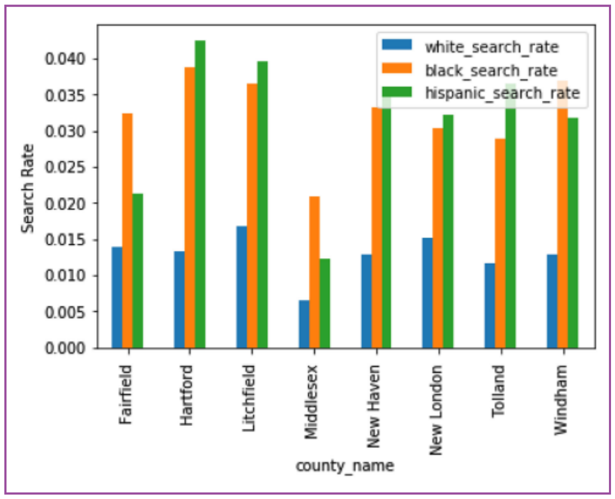

Figure 3

Feature bias localization in the *Stanford Open Policing* dataset for Connecticut, with county name as the feature and race as the protected attribute. In Hartford County, the ratio of search rates for the unprivileged groups (Black and Hispanic) in proportion to the search rate for the privileged group (this ratio is the DI fairness metric) is higher than the same metric in Middlesex County and others. The approach would localize Hartford County.

8 Bias mitigation algorithms

Bias mitigation algorithms attempt to improve the fairness metrics by modifying the training data, the learning algorithm, or the predictions. These algorithm categories are known as pre-processing, in-processing, and post-processing, respectively [33].

8.1 Bias mitigation approaches

The bias mitigation algorithm categories are based on the location where these algorithms can intervene in a complete machine learning pipeline. If the algorithm is allowed to modify the training data, then the pre-processing can be used. If it is allowed to change the learning procedure for a machine learning model, then in-processing can be used. If the algorithm can only treat the learned model as a black box without any ability to modify the training data or learning algorithm, then only post-processing can be used. This is illustrated in Figure 1.

8.2 Algorithms

AIF360 currently contains nine bias mitigation algorithms that span these three categories. All the algorithms are implemented by inheriting from the `Transformer` class. Transformers are an abstraction for any process that acts on an instance of `Dataset` class and returns a new, modified `Dataset` object. This definition encompasses pre-processing, in-processing, and post-processing algorithms.

Pre-processing algorithms: Reweighing [13] generates weights for the training examples in each (group, label) combination differently to ensure fairness before classification. Optimized pre-processing [26] learns a probabilistic transformation that edits the features and labels in the data with group fairness, individual distortion, and data fidelity constraints and objectives. Learning fair representations [25] finds a latent representation that encodes the data well but obfuscates information about protected attributes. Disparate impact remover [16] edits feature values to increase group fairness while preserving rank-ordering within groups.

In-processing algorithms: Adversarial debiasing [34] learns a classifier to maximize prediction accuracy and simultaneously reduce an adversary's ability to determine the protected attribute from the predictions. This approach leads to a fair classifier as the predictions cannot carry any group discrimination information that the adversary can exploit. Prejudice remover [17] adds a discrimination-aware regularization term to the learning objective.

Post-processing algorithms: Equalized odds post-processing [35] solves a linear program to find probabilities with which to change output labels to optimize equalized odds. Calibrated equalized odds post-processing [36]

Table 1 Statistics on the Test Suite for AIF360.

Metrics	Statistics
Number of Unit Test cases	23 test cases in 13 modules
Code Coverage (Helper Files)	65%
Code Coverage (Algorithms)	58%

optimizes over calibrated classifier score outputs to find probabilities with which to change output labels with an equalized odds objective. Reject option classification [15] gives favorable outcomes to unprivileged groups and unfavorable outcomes to privileged groups in a confidence band around the decision boundary with the highest uncertainty.

9 Maintaining code quality

Establishing and maintaining high-quality code is crucial for an evolving open-source system. Although we do not claim any novelty over modern software projects, we do feel that faithfully adopting such practices is another distinguishing feature of AIF360 relative to other fairness projects.

An extensible toolkit should provide confidence to its contributors that, while making it is easy for them to extend, it does not alter the existing API contract. Our testing protocols are designed to cover software engineering aspects and comprehensive test suites that focus on the performance metrics of fairness detection and mitigation algorithms.

The AIF360 Github repository is directly integrated with Travis CI,[6] a continuous testing and integration framework, which invokes `pytest` to run our unit tests. Any pull request automatically triggers the tests. The results of the tests are made available to the reviewer of the pull request to help ensure that changes to the code base do not introduce bugs that would break the tests.

Unit test cases ensure that classes and functions defined in the different libraries are functionally correct and do not break the flow of the fairness detection and mitigation pipeline. Each of our classes is equipped with unit tests that attempt to cover every aspect of the class/module/functions.

We have also developed a test suite to compute the metrics reported in Section 6. Our measurements include aspects of the fairness metrics, classification metrics, dataset metrics, and distortion metrics, covering a total of 71 metrics at the time of this writing. These metrics tests can be invoked directly with any fairness algorithm. The test suite also provides unit tests for all bias mitigation algorithms and basic validation of the datasets.

[6]https://travis-ci.org/

Our repository has two types of tests: 1) unit tests that test individual helper functions and 2) integration tests that test a complete flow of bias mitigation algorithms in Jupyter notebooks. **Table** provides the statistics and code coverage information as reported by the tool `py.test--cov` and Jupyter notebook coverage using `py.test--nbval`.

10 Web application

AIF360 includes not only the main toolkit code, but also an interactive Web experience (see Appendix C for a screen shot). Here, we describe its front-end and back-end design.

10.1 Design of the interactive experience

The Web experience was designed to provide useful information for diverse consumers. For business users, the interactive demonstration offers a sample of the toolkit's fairness checking and mitigation capabilities without requiring any programming knowledge. For new developers, the demonstration, notebook-based tutorials, guidance on algorithm selection, and access to a user community provide multiple ways to progress from their current level of understanding into the details of the code. For more advanced developers, the detailed documentation and code are directly accessible.

The design of the Web experience proceeded through several iterations. Early clickable mock-ups of the interactive demonstration had users first select a dataset, one or two protected attributes to check for bias, and one of up to five metrics to use for checking. We learned, however, that this was overwhelming, even for those familiar with AI, since it required choices they were not yet equipped to make. As a result, we simplified the experience by asking users to first select one of three datasets to explore. Bias-checking results were then graphically presented for two protected attributes across five different metrics. Users could then select a mitigation algorithm leading to a report comparing bias before and after mitigation.

Figure 4 shows two graphs from the interactive Web experience for one metric, disparate impact, before and after bias mitigation. To simplify interpretability, we used red to indicate the presence of above-threshold bias (the before case) and gray to indicate that bias had been adequately mitigated (the after case). These graphs also sought to give a sense of the distance between the unmitigated and mitigated cases. The information icon could be clicked to obtain additional information on the metric. Not all cases of bias could be adequately mitigated (see the Disparate Impact graph for another dataset in Appendix C).

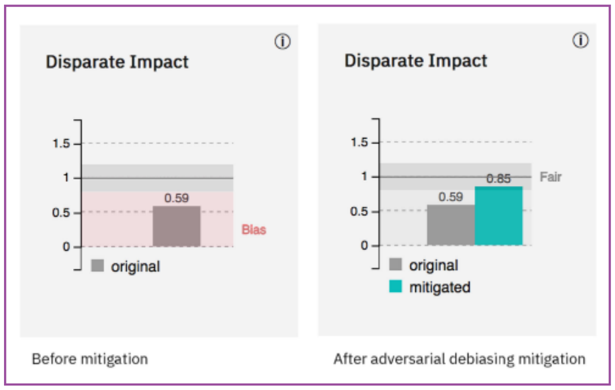

Figure 4

Graphs from the interactive Web experience showing one of the metrics, for one of the datasets, before and after mitigation.

The design of the rest of the site also went through several iterations. Of particular concern, the front page sought to convey toolkit richness while still being approachable. In the final design, a short textual introduction to the content of the site, along with direct links to the API documentation and code repository, is followed by a number of direct links to various levels of advice and examples. Further links to the individual datasets, the bias checkers, and the mitigation algorithms are also provided. In all this, we ensured the site was suitably responsive across all major desktop and mobile platforms.

10.2 Design of the back end service

The demo Web application not only provides a gentle introduction to the capabilities of the toolkit, but also serves as an example of deploying the toolkit to the cloud and converting it into a Web service. We used Python's Flask framework for building the service and exposed a REST API that generates a bias report based on the following input parameters from a user: the dataset name, the protected attributes, the privileged and unprivileged groups, the chosen fairness metrics, and the chosen mitigation algorithm, if any. With these inputs, the back end then runs a series of steps to: 1) split the dataset into training, development, and validation sets; 2) train a logistic regression classifier on the training set; 3) run the bias checking metrics on the classifier against the test dataset; and 4) if a mitigation algorithm is chosen, run the mitigation algorithm with the appropriate pipeline (pre-processing, in-processing, or post-processing). The end result is then cached so that if the exact same inputs are provided, the result can be directly retrieved from cache and no additional computation is needed.

The reason to truly use the toolkit code in serving the Web application rather than having a pre-computed lookup table of results is twofold: 1) we want to make the application a real representation of the underlying capabilities (in fact, creating the Web application helped us debug a few items in the code); and 2) we also avoid any issues of synchronizing updates to the metrics, explainers, and algorithms with the results shown: Synchronization is automatic. Currently, the service is limited to three built-in datasets, but it can be expanded to support the user's own data upload. The service is also limited to building logistic regression classifiers, but again this can be expanded. Such expansions can be more easily implemented if this fairness service is integrated into a full AI suite that provides various classifier options and data storage solutions.

11 Conclusion

AIF360 is an open-source toolkit that brings value to diverse users and practitioners. For fairness researchers, it provides a platform that enables them to 1) experiment with and compare various existing bias detection and mitigation algorithms in a common framework and gain insights into their practical usage; 2) contribute and benchmark new algorithms; 3) contribute new datasets and analyze them for bias. For developers, it provides 1) education on the important issues in bias checking and mitigation; 2) guidance on which metrics and mitigation algorithms to use; 3) tutorials and sample notebooks that demonstrate bias mitigation in different industry settings; and 4) a Python package for detecting and mitigating bias in their workflows.

Fairness is a multifaceted, context-dependent social construct that defies simple definition. The metrics and algorithms in AIF360 may be viewed through the lens of distributive justice [37], i.e., relating to decisions about who in a society receives which benefits and goods, and clearly do not capture the full scope of fairness in all situations. Even within distributive justice, more work is needed to apply the toolkit to additional datasets and situations. Future work could also expand the toolkit to measure and mitigate other aspects of justice such as compensatory justice, i.e., relating to the extent to which people are fairly compensated for harm done to them. Further work is also needed to extend the variety of types of explanations offered and to create guidance for practitioners on when a specific kind of explanation is most appropriate. There is a lot of work left to do to achieve unbiased AI. We hope others in the research community continue to contribute their own approaches to fairness and bias checking, mitigation, and explanation to the toolkit.

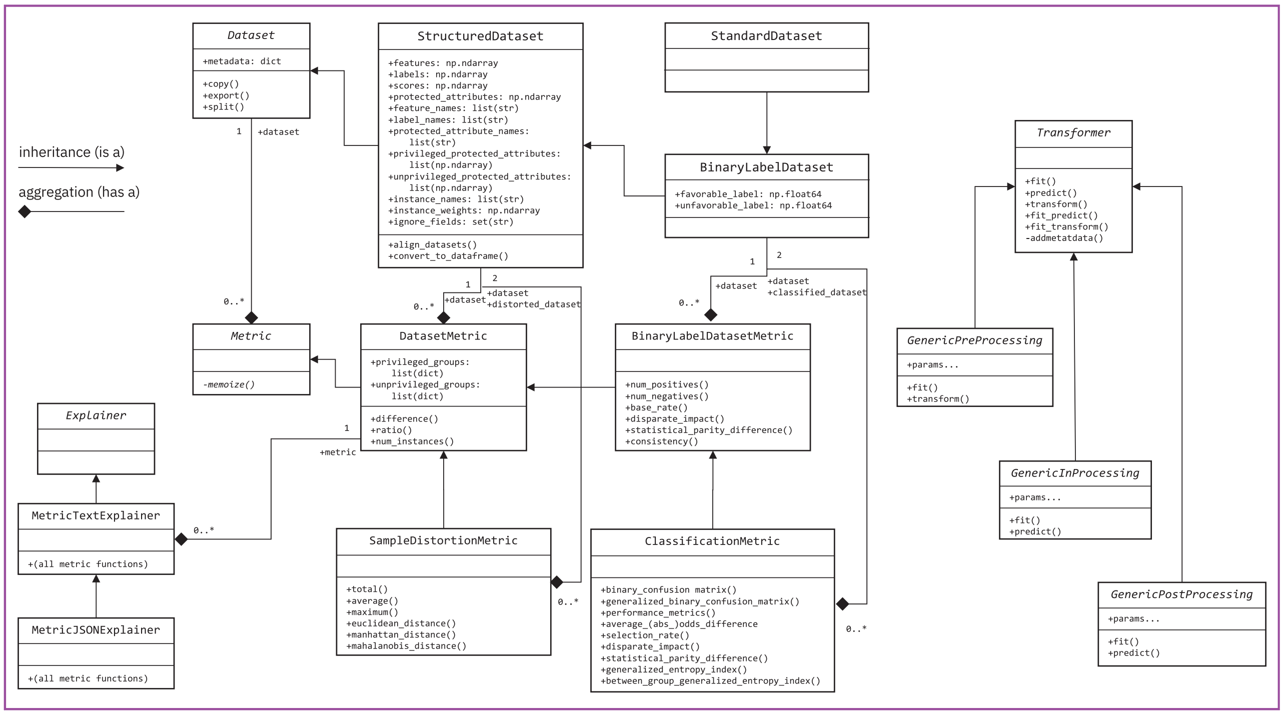

Figure 5

Class abstractions for a fair machine learning pipeline, as implemented in AIF360.

Appendix A: UML class diagram

Figure 5 shows class abstractions for a fair machine learning pipeline, as implemented in AIF360. This figure is meant to provide a visual sense of the class hierarchy, so many details and some methods are omitted. For brevity, inherited members and methods are not shown (but overridden ones are), nor are aliases such as `recall()` for `true_positive_rate()`. Some methods are "meta-metrics"—such as `difference()`, `ratio()`, `total()`, `average()`, `maximum()`—that act on other metrics to get, e.g., `true_positive_rate_difference()`. The metric explainer classes use the same method signatures as the metric classes (not enumerated) but provide further description for the values. The `GenericPreProcessing`, `GenericInProcessing`, and `GenericPostProcessing` are not actual classes but serve as placeholders here for the real bias mitigation algorithms we implemented. Finally, `memoize` and `addmetadata` are Python decorator functions that are automatically applied to every function in their respective classes.

Appendix B: Code Snippets

This example provides Python code snippets for some common tasks that the user might perform using our toolkit. The example involves the user loading a dataset, splitting it into training and testing partitions, understanding the outcome disparity between two demographic groups, and transforming the dataset to mitigate this disparity. Several other examples and tutorials for measuring and mitigating bias are available at https://github.com/IBM/AIF360/tree/master/examples

Dataset operations

```
 Load the UCI Adult dataset
from aif360.datasets import AdultDataset
ds_orig = AdultDataset()

 Split into train and test partitions
ds_orig_tr, ds_orig_te = ds_orig.split([0.7], shuffle=True, seed=1)
```

```
  Look into the training dataset
print(''Training Dataset shape'')
print(ds_orig_tr.features.shape)
print(''Favorable and unfavorable outcome labels'')
print(ds_orig_tr.favorable_label, ds_orig_tr.unfavorable_label)
print(''Metadata for labels'')
print(ds_orig_tr.metadata[''label_maps''])
print(''Protected attribute names'')
print(ds_orig_tr.protected_attribute_names)
print(''Privileged and unprivileged protected attribute values'')
print(ds_orig_tr.privileged_protected_attributes,
  ds_orig_tr.unprivileged_protected_attributes)
print(''Metadata for protected attributes'')
print(ds_orig_tr.metadata[''protected_attribute_maps''])
```

Expected output: The attributes of the `Adult` dataset will be printed. The training partition of the *Adult* dataset has 31,655 instances and 98 features with two protected attributes (`race` and `sex`). The labels correspond to high-income (>5 K) or low-income ($=5$ K), as shown in the metadata. Similar metadata is also available for protected attributes.

Checking for bias in the original data

```
  Load the metric class
from aif360.metrics import BinaryLabelDatasetMetric

  Define privileged and unprivileged groups
priv = [{'sex': 1}]  Male
unpriv = [{'sex': 0}]  Female

  Create the metric object
metric_otr = BinaryLabelDatasetMetric( ds_orig_tr,
  unprivileged_groups=unpriv, privileged_groups=priv)

  Load and create explainers
from aif360.explainers import MetricTextExplainer, MetricJSONExplainer
text_exp_otr = MetricTextExplainer(metric_otr)
json_exp_otr = MetricJSONExplainer(metric_otr)

  Print statistical parity difference
print(text_exp_otr.statistical_
parity_difference())
print(json_exp_otr.statistical_
parity_difference())
```

Expected output: The statistical parity difference should be -0.1974, which is the difference between probability of favorable outcome (high income) between the unprivileged group (females) and the privileged group (male) in this dataset. The JSON output is more elaborate to facilitate consumption by a downstream algorithm.

Pre process data to mitigate bias

```
  Import the reweighing preprocessing algorithm class
from aif360.algorithms.preprocessing.reweighing import Reweighing

  Create the algorithm object
RW = Reweighing(unprivileged_
groups=unpriv, privileged_groups=priv)
```

```
  Train and predict on the training data
  Uses scikit-learn convention (fit, predict, transform)
RW.fit(ds_orig_tr)
ds_transf_tr = RW.transform(ds_orig_tr)
```

Expected output: There will be no output here, but the reweighing algorithm equalizes the weights across (group, label) combination.

Checking for bias in the pre processed training data

```
 Create the metric object for pre-processed data
metric_ttr = BinaryLabelDatasetMetric(ds_transf_tr,
  unprivileged_groups=unpriv, privileged_groups=priv)

  Create explainer
text_exp_ttr = MetricTextExplainer(metric_ttr)

  Print statistical parity difference
print(text_exp_ttr.statistical_parity_difference())
```

Expected output: Because of the action of the re-weighing pre-processing algorithm, the statistical parity difference for the transformed data (`ds_transf_tr`) must be really close to 0.

Pre process out of sample testing data and check for bias

```
  Apply the learned re-weighing pre-processor
ds_transf_te = RW.transform(ds_orig_te)
  Create metric objects for original and
  pre-processed test data
metric_ote = BinaryLabelDatasetMetric(ds_orig_te,
  unprivileged_groups=unpriv, privileged_groups=priv)
metric_tte = BinaryLabelDatasetMetric(ds_transf_te,
  unprivileged_groups=unpriv, privileged_groups=priv)

  Create explainers for both metric objects
text_exp_ote = MetricTextExplainer(metric_ote)
text_exp_tte = MetricTextExplainer(metric_tte)

  Print statistical parity difference
print(text_exp_ote.statistical_parity_difference())
print(text_exp_tte.statistical_parity_difference())
```

Expected output: The trained reweighing pre-processor can be applied on the out-of-sample test data. The metrics for the original and transformed testing data will show a significant reduction in statistical parity difference (-0.2021 to -0.0119 in this case).

Appendix C: UI page

Figure 6 is a screen shot from the Web interactive experience, showing the results of mitigation applied to one of the available datasets.