Fairness specification and repair for machine learning pipeline

by

Giang Nguyen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Co-major Professor
Wei Le, Co-major Professor
Ying Cai
Gurpur Prabhu
Qi Li
Simanta Mitra

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2024

DEDICATION

I would like to dedicate this thesis to my family without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vi

LIST OF FIGURES . . . viii

ACKNOWLEDGMENTS . . . ix

ABSTRACT . . . xi

CHAPTER 1. GENERAL INTRODUCTION . . . 1
- 1.1 Contribution . . . 4
- 1.2 Background & Related Work . . . 5
  - 1.2.1 Fairness in ML Pipeline . . . 5
  - 1.2.2 Software Engineering for ML Fairness . . . 9
- Bibliography . . . 10

CHAPTER 2. FIX FAIRNESS, DON'T RUIN ACCURACY: PERFORMANCE AWARE FAIRNESS REPAIR USING AUTOML . . . 18
- 2.1 Abstract . . . 18
- 2.2 Introduction . . . 19
- 2.3 Background . . . 21
  - 2.3.1 Preliminaries . . . 21
  - 2.3.2 Related Work . . . 23
- 2.4 Motivation . . . 24
- 2.5 Problem Definition . . . 27
- 2.6 Fair-AutoML . . . 29
  - 2.6.1 Dynamic Optimization for Bias Elimination . . . 29
  - 2.6.2 Search Space Pruning for Efficient Bias Elimination . . . 32
- 2.7 Evaluation . . . 36
  - 2.7.1 Experiment . . . 37
  - 2.7.2 Effectiveness (RQ1) . . . 39
  - 2.7.3 Adaptability (RQ2) . . . 42
  - 2.7.4 Ablation Study (RQ3) . . . 45
- 2.8 Discussion . . . 47
- 2.9 Threats to Validity . . . 48
- 2.10 Conclusion . . . 49
- Bibliography . . . 49

CHAPTER 3. DESIGN BY FAIRNESS CONTRACT FOR MACHINE LEARNING PIPELINE 58
3.1 Abstract . . . 58
3.2 Introduction . . . 59
3.3 Motivation . . . 61
3.4 Problem Formulation . . . 63
3.4.1 Classification Program . . . 63
3.4.2 ML Pipeline . . . 63
3.4.3 Fairness Specification . . . 64
3.5 Guidelines for Writing, Collecting, and Using Fairness Contracts . . . 67
3.5.1 Writing Fairness Contracts . . . 67
3.5.2 Collecting Fairness Contracts . . . 68
3.5.3 Using Fairness Contracts . . . 68
3.6 Fairness Contracts . . . 70
3.6.1 Global Fairness Contracts . . . 71
3.6.2 Decomposing Global Fairness Contract into Modules . . . 73
3.6.3 Fairness Contract Runtime Assertion Technique . . . 75
3.7 Evaluation . . . 76
3.7.1 Experiment Setup . . . 76
3.7.2 Effectiveness (RQ1) . . . 78
3.7.3 Efficiency (RQ2) . . . 80
3.7.4 Applicability (RQ3) . . . 83
3.8 Discussion . . . 83
3.8.1 Limitations . . . 83
3.8.2 Threats to Validity . . . 84
3.9 Conclusions . . . 84
Bibliography . . . 84

CHAPTER 4. MODULAR FAIRNESS CHECKER FOR MACHINE LEARNING PIPELINE 94
4.1 Abstract . . . 94
4.2 Introduction . . . 95
4.3 Motivation . . . 99
4.4 Background . . . 101
4.4.1 Concentration Inequalities . . . 101
4.4.2 Sensitivity . . . 101
4.4.3 Related Work . . . 102
4.5 Problem Formulation . . . 103
4.5.1 Modular Fairness Checker Inputs . . . 103
4.5.2 Modular Fairness Checker Definition . . . 104
4.5.3 Modular Fairness Specification . . . 105
4.6 Modular Fairness Checker . . . 106
4.6.1 Modular Fairness Specification Syntax . . . 106
4.6.2 Modular Fairness Checking using Concentration Inequality . . . 109
4.6.3 Fairness-Aware Feature Importance for Modules . . . 111
4.7 Evaluation . . . 114
4.7.1 Experimental Setup . . . 115
4.7.2 Effectiveness of Fairness Bug Detection (RQ1) . . . 116

4.7.3 Efficiency of the Concentration Inequalities (RQ2) 119
4.7.4 Fairness-aware Feature Importance (RQ3) 120
4.7.5 Generalization (RQ4) 122
4.8 Threats to Validity 123
4.9 Conclusion 124
Bibliography 125

CHAPTER 5. GENERAL CONCLUSION 135
5.1 Future Work 136
5.1.1 Fairness Bugs Repairing for Deep Learning (DL) Systems 136
5.1.2 Fairness Bug Localization for Deep Learning Models 136
5.1.3 Balancing Modular and Global Fairness in Machine Learning Pipelines 137
Bibliography 138

LIST OF TABLES

Page

Table 2.1 Mean proportions of mitigation cases that fall into each mitigation region . 27

Table 2.2 Trade-off assessment results of *Fair-AutoML*, *Auto-Sklearn*, and mitigation techniques 40

Table 2.3 Fair-AutoML (FA) vs bias mitigation methods in fixing fairness bugs 41

Table 2.4 Proportion of *Fair-AutoML*, *Auto-Sklearn*, and mitigation techniques that fall into each mitigation region 43

Table 2.5 Accuracy and fairness achieved by Fair-AutoML and Pafait-ML on Pafait-ML's benchmark 44

Table 2.6 Trade-off assessment results of *Auto-Sklearn*, *FAv1*, and *FAv2* 46

Table 3.1 Fairness Contracts 69

Table 3.2 The datasets used in the fairness bug detection experimentation. 77

Table 3.3 Comparison of number of detected bugs between *Fairness Contract* and fairness bug fixing techniques 79

Table 3.4 Effectiveness of *Fairness Contract* 80

Table 3.5 Comparison of runtime between *Fairness Contract* and fairness repairing methods on Adult Census (AC) 81

Table 3.6 Comparison of runtime between *Fairness Contract* and fairness repairing methods on Bank Marketing (BM) 81

Table 3.7 Comparison of runtime between *Fairness Contract* and fairness repairing methods on German Credit (GC) 81

Table 3.8 Comparison of runtime between *Fairness Contract* and fairness repairing methods on Titanic (TT) 81

Table 4.1 Local fairness of stages as downstream transformer. [17] 99

Table 4.2 Average proportion of accurately detected bugs by fairness bug detection techniques. 100

Table 4.3 Effectiveness of concentration inequalities 119

Table 4.4 Effectiveness of fairness-aware feature importance on Adult Census (AC) . . 121

Table 4.5 Effectiveness of fairness-aware feature importance on Bank Marketing (BM) 121

Table 4.6 Effectiveness of fairness-aware feature importance on German Credit (GC) . 122

Table 4.7 Effectiveness of fairness-aware feature importance on Titanic (TT) 122

LIST OF FIGURES

Page

Figure 2.1 Baseline fairness-accuracy trade-off [41] 26

Figure 2.2 An Overview of *Fair-AutoML* Approach 29

Figure 2.3 Accuracy and fairness achieved by Fair-AutoML (green circle) and Pafait-ML (orange circle) with Decision Tree (left) and logistic regression (right) on Adult dataset (Pafait-ML's benchmark). The blue line shows the Fairea baseline and red lines define the trade-off regions. 45

Figure 3.1 Motivating Example for Fairness Contracts 62

Figure 3.2 Comparison of average runtime 82

Figure 4.1 ML Pipeline with data preprocessing modules, each having individual fairness impact. 95

Figure 4.2 Specification syntax. Here, S and T are nonterminal symbols, with S serving as the start symbol. The symbol $\langle E \rangle$ represents expressions of random variables. 106

Figure 4.3 Comparison of detected bugs between *Fairness Checker* (FC) and fairness detection techniques. 117

Figure 4.4 Average runtime of bias detection tools 118

Figure 4.5 Average runtime of FA and FP 123

ACKNOWLEDGMENTS

I would like to take this opportunity to acknowledge everyone who has helped me with various aspects of conducting research and writing this dissertation. First and foremost, I would like to express my deepest gratitude to Dr. Hridesh Rajan for his supervision, patience, and continuous support throughout this research journey. I am thankful to the US National Science Foundation under Grant No. CCF-15-13263, Grant No. CCF-19-34884, Grant No. CNS-21-20448, and Facebook Probability and Programming Award under Grant No. 809725759507969 for financially supporting this project. I extend my heartfelt thanks to my other committee members, Dr. Wei Le, Dr. Ying Cai, Dr. Gurpur Prabhu, Dr. Qi Li, and Dr. Simanta Mitra, for their invaluable guidance and suggestions. My sincere thanks go to all the members of the Laboratory for Software Design, whose constructive feedback and timely suggestions have been instrumental during my research. Lastly, I am profoundly grateful to my family. Their continuous love, support, and gratitude have been a constant source of strength throughout my PhD journey.

The majority of this draft is adopted from the previous peer-reviewed papers published and submitted in top-tier software engineering venues. Chapter 2 is based on our paper published in the proceedings of the 31st ACM Joint Meeting on European Software Engineering (ESEC/FSE '23). Chapter 3 is based on the paper submitted in the proceedings of the 47th International Conference on Software Engineering (ICSE '25). Chapter 4 is based on the paper submitted in the proceedings of the 33rd ACM international conference on the foundations of software engineering (FSE '25). Furthermore, I supported open science as I continuously published code, data, and benchmarks as peer-reviewed artifacts. In this dissertation, I do not include some of the other research works that I conducted during my Ph.D., e.g., leveraging hand-crafted models from open-source repositories to optimize neural architecture search, studying deep learning model bugs and bug-fix patterns, proposing an approach to detect and fix API bugs in real-world deep learning programs. For the presentation of this dissertation, ChatGPT was utilized to refine and enhance its clarity and structure.

ABSTRACT

Machine learning (ML) algorithms are increasingly used in critical decision-making systems, such as criminal sentencing, employee hiring, bank loan approvals, and college admissions, all of which have a direct impact on human lives. The fairness of these ML-based systems has become a significant concern in recent years. Numerous incidents have highlighted instances where ML models have discriminated against individuals based on protected attributes like race, gender, age, and religious beliefs. While considerable research has been devoted to identifying and mitigating unfairness in ML models, these methods lack generalizability as they only perform effectively in specific datasets, ML algorithms, or fairness metrics. This dissertation introduces novel techniques for **optimizing fairness** within machine learning pipelines, addressing the limitations of existing bias mitigation approaches. In the first approach, we introduce *Fair-AutoML*, the first generalized method for addressing fairness bugs (optimizing fairness) using AutoML. Unlike existing bias mitigation techniques, *Fair-AutoML* addresses their limitations by enabling efficient and fairness-aware Bayesian search to repair unfair models, making it effective for a wide range of datasets, models, and fairness metrics. There are two innovative contributions of *Fair-AutoML*, including a dynamic optimization function and a fairness-aware search space. By dynamically adjusting the optimization function to balance accuracy and fairness, we can effectively reduce bias with little to no impact on accuracy. Moreover, our fairness-aware search space pruning method allows *Fair-AutoML* to enhance fairness with minimal computational cost and repair time.

In machine learning tasks, it is standard practice to construct a pipeline comprising an ordered set of stages, including data acquisition, preprocessing, modeling, and more. However, designing ML pipelines that adhere to fairness standards is challenging due to their inherent complexity. Various components, such as data preprocessing and feature engineering, play a crucial role in determining the system's overall fairness. While numerous bias mitigation and testing techniques have been developed to enhance fairness in ML pipelines, they often fail to detect and explain the root causes of bias effectively. We propose two novel methods, *Fairness Contract* and *Fairness Checker*, which apply design-by-contract (DbC) principles to ensure algorithmic fairness within the ML pipeline, enabling the **detection** and **explanation** of bias within the ML pipeline. First, we introduce a preventive mechanism called ”*Fairness Contract*,” which can detect fairness violations in real-time as the ML program executes. Its modular design enables us to identify the precise location of fairness violations within the ML software. Second, we introduce a novel approach, *Fairness Checker*, which uses concentration inequalities to identify bias in the ML pipeline. Our method starts by calculating fairness scores for the dataset features, revealing which features are positively or negatively affected after the pipeline's preprocessing steps. These findings are then applied to concentration inequalities to uncover bias issues within the ML pipeline. Furthermore, we created a fairness annotator that enables developers to specify and enforce modular fairness throughout the pipeline's development process.

CHAPTER 1. GENERAL INTRODUCTION

ML-based software is increasingly being used in applications that directly impact human lives, such as loan approval, criminal sentencing, and employee hiring [12, 10, 32]. A major concern in these contexts is whether we can trust the ML pipeline to make fair decisions. Designing ML pipelines that meet fairness standards is challenging due to their inherent complexity and the variety of components involved. Additionally, fairness can be defined in multiple ways, and different models or systems may require distinct fairness metrics for proper evaluation. Therefore, developing a fair ML pipeline requires a deep understanding of the characteristics of diverse ML algorithms, data properties, and fairness metrics. This dissertation aims to address this problem by proposing solutions to optimize fairness and detect bias at various stages of ML pipelines, generalizing across diverse algorithms, datasets, and fairness metrics.

In recent years, significant research has been conducted to address unfairness in machine learning. However, software engineers still require effective and scalable tools to measure, localize, test, verify, repair, and explain fairness issues in practice [9, 25, 4, 26, 44, 15, 27, 45]. While algorithms have been developed to measure fairness [31, 20, 19] and mitigate bias [13, 17, 23, 31, 36], they often fall short of providing generalize solutions. First, their effectiveness varies depending on the ML algorithms, datasets, and the selection of fairness metrics [22, 23, 31, 16]. Moreover, current bias mitigation techniques frequently result in reduced accuracy. Fairea [34], an innovative method for assessing the effectiveness of bias mitigation strategies, reveals that almost half of the cases evaluated showed poor effectiveness. Second, existing bias mitigation and testing methods are limited in their ability to localize the source of bias. They primarily focus on detecting whether the ML pipeline is biased rather than identifying the root cause of that bias. This shortcoming leaves developers with limited insights into how and why bias has entered the system, making it challenging to implement targeted and effective solutions. Third, current bias explanation methods [8, 29, 33] are not able to automatically explain the sources of bias within machine learning models. They do not offer a systematic or automated way to explain how bias arises in the ML pipeline. In this dissertation, we propose novel solutions to address these limitations of existing methods.

To overcome the limitations in the generalization of existing bias mitigation techniques, we introduce *Fair-AutoML*, a novel approach that leverages AutoML to address fairness issues in ML models. Our proposed solution is a novel approach that utilizes automated machine learning (AutoML) techniques to mitigate bias. Our approach includes two key innovations: a novel optimization function and a fairness-aware search space. By improving the default optimization function of AutoML and incorporating fairness objectives, we are able to mitigate bias with little to no loss of accuracy. Additionally, we propose a fairness-aware search space pruning method for AutoML to reduce computational cost and repair time. Our approach, built on the state-of-the-art *Auto-Sklearn* tool, is designed to reduce bias in real-world scenarios. In order to demonstrate the effectiveness of our approach, we evaluated our approach on four fairness problems and 16 different ML models, and our results show a significant improvement over the baseline and existing bias mitigation techniques. Our approach, *Fair-AutoML*, successfully repaired 60 out of 64 buggy cases, while existing bias mitigation techniques only repaired up to 44 out of 64 cases.

Our previous work, *Fair-AutoML*, effectively mitigates bias without compromising accuracy and can generalize across various ML algorithms, datasets, and fairness metrics. However, *Fair-AutoML* primarily enhances the fairness of ML pipelines by tuning model hyperparameters. This means that if the source of bias lies in a component outside the model itself, *Fair-AutoML* may not be able to address it. To overcome the limitations of *Fair-AutoML*, as well as those of existing bias mitigation and testing methods in localizing bias within the ML pipeline, we propose a preventive measure called a "*Fairness Contract*." This approach integrates design-by-contract (DbC) principles for algorithmic fairness directly into the ML pipeline. Although *Fairness Contract* builds upon traditional DbC methods, we must address unique challenges. First, two types of fairness properties, hyperproperties and probabilistic properties, cannot be specified using conventional

DbC mechanisms. Second, traditional DbC techniques do not support runtime assertion checking to report fairness violations in the ML pipeline. We address these issues by introducing a contract mechanism to capture fairness properties at different ML pipeline stages and report bias issues during runtime. These contracts abstract away the details of the ML algorithms, enabling systematic analysis and verification. In this work, we designed 24 contracts specifically targeting fairness bugs at various stages of the ML pipeline. These contracts were also used to evaluate our method on four fairness tasks, 45 buggy codes, and 24 correct codes. Our approach, *Fairness Contract*, is able to localize fairness bugs during runtime, which existing bug repair techniques cannot achieve. Moreover, *Fairness Contract* successfully identified 40 out of 45 buggy cases, outperforming existing fairness bug-fixing methods, which only managed to fix up to 35 out of 45 bugs. *Fairness Contract* also efficiently detects fairness bugs across various ML algorithms with a significantly shorter runtime. It only takes 1.5 seconds to detect and locate a bug, compared to the fastest existing method, which requires up to 30.9 seconds.

Fairness Contract is designed to describe the fairness behavior of individual components within the ML pipeline in isolation. This feature provides a detailed understanding of the impact on fairness and accuracy when changes are made within the pipeline. As a result, *Fairness Contract* can instantly localize fairness violations as they occur during the execution of the ML program. However, *Fairness Contract* does not measure the extent to how much fairness can be improved after applying ML algorithms to the pipeline. In other words, it cannot quantify the potential increase in fairness values once an ML algorithm is implemented in an ML pipeline, leaving some uncertainty regarding the effectiveness of the algorithm. To address this limitation, we propose *Fairness Checker*, an extension of *Fairness Contract*, to detect fairness bugs at the modular level and explain their root causes. Our method begins by computing fairness scores for the dataset features, highlighting which features are positively or negatively impacted after applying preprocessing components in the pipeline. These insights are then applied to concentration inequalities to detect the fairness bugs within the ML pipeline. Additionally, we developed a programming language abstraction that allows developers to specify and enforce modular fairness during ML pipeline development. We evaluated our approach using four datasets, two fairness metrics, and 13 ML algorithms. *Fairness Contract* has a high success rate in detecting fairness bugs. Specifically, our method identified 42 out of 45 buggy cases with an average processing time of 26 seconds per case, outperforming existing bias detection methods, which identified up to 35 out of 45 cases with an average of 31 seconds per case.

1.1 Contribution

In this dissertation, first, we optimize the fairness within the real-world ML pipeline. To achieve this goal, we tuned the hyperparameter to fix fairness performance bugs without losing accuracy. The approach is able to fix fairness across diverse datasets, fairness metrics, and ML algorithms. However, merely optimizing for fairness is not sufficient, given the inherent complexity of real-world ML pipelines. These pipelines often involve numerous stages, such as data collection, preprocessing, feature engineering, model selection, and evaluation. Bias can be introduced at any of these stages, making it crucial for developers to not only address fairness at the model level but also to gain a comprehensive understanding of where and why bias occurs. Without this insight into the sources and underlying causes of bias, efforts to mitigate it may be incomplete or ineffective. Therefore, achieving fairness in ML systems requires a holistic approach that involves detecting and explaining each component of the pipeline to identify potential biases and understand their origins. This deeper understanding empowers developers to implement more targeted and effective strategies for bias mitigation, ultimately leading to more equitable outcomes. As a solution to the problem, we are the first to propose design-by-contracts approaches to specifying fairness properties within the ML pipeline to detect and explain bias.

In Chapter 2, we have proposed a novel approach to address unfairness bugs while retaining model accuracy [39]. Our method automatically generates the optimization function based on the input, making the AutoML process for fixing fairness issues more efficient. Additionally, we have introduced an automatic pruning of the search space based on input data, enabling faster resolution of fairness bugs using AutoML. We have implemented this approach in a state-of-the-art AutoML system, *Auto-Sklearn* [24], and the resulting artifact is available here [28].

In Chapter 3, we are the first to propose the use of fairness contracts for ML pipelines by utilizing preconditions and postconditions to specify fairness properties. To facilitate this, we introduced a fairness specification mechanism that abstracts both algorithm characteristics and data properties. Additionally, we designed and contributed 24 fairness contracts capable of detecting and localizing fairness bugs within ML pipelines. Furthermore, we developed the *Fairness Contract* [2] framework, which is extensible to a wide range of ML algorithm types, providing a versatile tool for ensuring fairness in machine learning systems.

In Chapter 4, we are the first to design fairness annotators to specify fairness properties during the implementation phase. Additionally, we proposed a novel fairness-aware feature importance technique to explain *modular fairness*, providing deeper insights into model fairness. To enhance the detection of *modular fairness* bugs, we introduced a new method based on concentration inequalities. Furthermore, we developed the *Fairness Checker* [1] framework, which is extensible to a wide variety of ML algorithm types, offering a comprehensive approach to ensuring fairness across different models.

1.2 Background & Related Work

1.2.1 Fairness in ML Pipeline

Fairness in machine learning has become a critical area of research due to the potential for ML models to perpetuate or even exacerbate existing societal biases. This concern has prompted extensive research on identifying, measuring, and mitigating biases throughout the ML pipeline, from data collection to model deployment. Dwork and Ilvento argued that fairness is dynamic in environments with multiple components [21]. They explored how fairness is composed in multi-classification settings, introducing the theory of ”functional composition.” This theory involves combining the binary outputs of multiple classifiers using logical operators such as AND and OR. Bower *et al.* examined fairness within the ML pipeline, viewing the pipeline as a sequence of multiple classification tasks [11]. They demonstrated that even when individual components are fair, the compounded final decision may still be unfair. This was illustrated using the equal opportunity definition of fairness and a two-stage hiring pipeline where candidates move from one stage to the next. Grgic-Hlaca *et al.* discussed procedural fairness concerning different dataset features [30]. D'Amour *et al.* studied the dynamics of fairness in multi-classification environments through simulations [18]. More recently, Yang *et al.* [47] proposed software instrumentation in the ML pipeline, while [49] introduced a methodology to detect the impact of preprocessing stages in the ML pipeline.

Fairness Metrics. The machine learning community has established numerous fairness criteria and introduced various metrics to assess fairness in classification tasks [48, 19, 40, 20, 23, 31, 13, 50, 17, 42]. Different metrics are suited to particular contexts and reflect specific views of fairness. These metrics are generally divided into two main categories: 1) group fairness and 2) individual fairness. Group fairness ensures that protected groups (e.g., male versus female, young versus old) receive similar treatment in the model's predictions. Conversely, individual fairness requires that two similar individuals, differing only in their protected attribute, are treated similarly in the model's predictions [26, 35]. Galhotra *et al.* pointed out that group fairness might fail to identify bias in cases where discrimination occurs equally across groups [26], prompting the adoption of individual fairness in recent studies. [4, 26, 53]. In this dissertation, both group and individual fairness criteria are considered as relevant. Specifically, we employed the most widely used fairness metrics from existing literature.

Data Collection and Preprocessing. The initial stage of the ML pipeline—data collection—plays a crucial role in determining the fairness of a model. Previous studies have highlighted how biased data can stem from historical inequities and skewed sampling processes [6]. For instance, datasets used in predictive policing may reflect and reinforce societal biases against certain demographic groups, leading to unfair outcomes in model predictions. To address these issues, a range of data *pre-processing* bias mitigators have been proposed. For instance, Fair-SMOTE [14] addresses data bias by removing biased labels and balancing the distribution of positive and negative examples for each sensitive attribute. Reweighing [36] decreases bias by assigning different weights to different groups based on the degree of favoritism of a group. Disparate Impact Remover [23] is a pre-processing bias mitigation technique that aims to reduce bias by editing feature values.

Fairness-Aware Learning Algorithms. While preprocessing methods aim to address biases in the input data, *in-processing* techniques integrate fairness considerations directly into the learning algorithms. These methods modify the learning process to enforce fairness constraints or penalize unfair behaviors. For example, Zafar *et al.* [48] proposed a fairness-aware logistic regression model that incorporates fairness constraints like disparate impact into the objective function. By optimizing this objective, the model is trained to minimize prediction disparities between different groups.

Another prominent approach in this category is adversarial learning for fairness. Inspired by the success of adversarial networks, Zhang *et al.* [51] introduced adversarial debiasing, where an adversary attempts to predict sensitive attributes (e.g., race, gender) from the model's predictions. The main model learns to make accurate predictions while simultaneously fooling the adversary, resulting in representations that are less informative of sensitive attributes. This approach has been shown to improve fairness without significantly sacrificing model performance.

Regularization techniques have also been explored to promote fairness during training. Agarwal *et al.* [3] proposed a reduction-based approach that translates fairness constraints into regularization terms within the loss function, penalizing the model for disparities in error rates across different demographic groups. Such techniques ensure that the model's predictions are equitable across sensitive subgroups.

Post-processing Techniques. *Post-processing* methods are employed after the model has been trained to adjust its predictions in a way that promotes fairness. Pleiss *et al.* [41] proposed calibrated equalized odds post-processing, which modifies the model's output to equalize false positive and false negative rates across different groups without retraining the model. Another approach involves threshold adjustment where decision thresholds are altered for different demographic groups to achieve desired fairness properties [31]. These post-processing techniques are often more practical in real-world settings, where modifying the data or the learning algorithm may be infeasible. However, they can also be limited by the degree to which the original model is biased, as they do not address underlying biases in the data or model training process.

Fairness in Deployment and Monitoring. While much research has focused on fairness up to the point of model deployment, ensuring fairness in the operational phase is equally important. Biases can emerge during deployment due to changing data distributions or feedback loops, where the model's predictions influence future data. Liu *et al.* [37] discussed the phenomenon of bias amplification, where the deployment of biased models can further skew the data collected over time, resulting in a feedback loop that exacerbates existing biases.

End-to-End Fairness in the ML Pipeline. Given that fairness issues can arise at multiple stages of the ML pipeline, recent work has begun to address fairness in an end-to-end manner. Veale *et al.* [46] argued for a holistic approach that considers fairness throughout the entire pipeline, from data collection and preprocessing to model training, evaluation, and deployment. They highlighted the importance of cross-disciplinary collaboration, combining technical solutions with insights from social sciences to ensure fair and ethical ML practices.

Despite significant advances, many challenges remain in achieving fairness in ML pipelines. One key issue is the existing methods are not generalized to a diverse range of ML algorithms and fairness metrics. Moreover, the trade-offs between fairness and other model objectives, such as accuracy, often necessitate careful consideration of societal values and priorities. In this dissertation, we aim to overcome the limitations of current bias mitigation methods by focusing on the fairness-accuracy trade-off and improving generalization.

1.2.2 Software Engineering for ML Fairness

Recently, the software engineering community has also focused on fairness in ML, mostly on fairness testing [9, 25, 4, 26, 44]. These works propose methods to generate appropriate test data inputs for the model and prediction on those inputs characterizes fairness. Some research has been conducted to generate general specification language that allows a programmer to specify a range of fairness definitions from the literature [5, 7, 52, 38]. In addition, empirical studies have been conducted to explain bias in ML pipelines [8, 29, 33].

Fairness Testing. SE researchers have developed various fairness testing methods to detect bias in ML software [26, 43, 20, 44, 4]. These works propose methods for generating suitable test data inputs, where the model's predictions on these inputs help to assess fairness. For instance, Themis [26] automatically generates random tests to identify causal fairness in a black-box decision-making process. FairTest [43] introduces a method for detecting unwarranted feature associations and potential biases in datasets through manually crafted tests. FairML [20] presents an orthogonal transformation method to measure the relative dependence of black-box models on their input features, with the aim of assessing fairness. Aequitas [44] offers a fully automated test generation module aimed at creating discriminatory inputs in ML models to validate individual fairness. A more recent study [4] proposes a black-box fairness testing approach for detecting individual discrimination in ML models, utilizing a test case generation algorithm based on symbolic execution and local explainability. While these works have introduced innovative techniques for detecting and testing fairness in ML systems, our focus has been on localizing and explaining the origin of bias.

Fairness Specification. Recent research has focused on specifying fairness in ML softwares [5, 7, 52, 38]; however, these works fall short in localizing fairness issues and demonstrating generalizability across different ML algorithms. For example, OmniFair [52] enables users to define group fairness constraints in machine learning declaratively. It supports all major group fairness constraints, including custom ones, and can handle multiple constraints at once.

Similarly, Albarghouthi *et al.* introduced fairness-aware programming [5], which allows fairness definitions to be declaratively specified within decision-making code and verified during runtime. Previous efforts have made significant progress in specifying fairness; however, these techniques focus solely on global fairness, such as group fairness or individual fairness, which does not help users locate specific problems. Our focus is to specify both global and modular fairness, demonstrating generalizability across various datasets and ML algorithms.

Bias Explanation. To address the challenge of bias, researchers have developed various techniques to explain and mitigate it in machine learning models. Explainability is crucial for understanding how and why a model makes biased decisions [8, 29, 33]. Biswas *et al.* [8] explain the impact of data preprocessing stages on the fairness of classification tasks. They introduced a causal method and utilized existing metrics to assess the fairness of these preprocessing steps. The findings revealed that many of these stages introduce bias into the predictions. By evaluating the fairness of these data transformers, it is possible to construct fairer machine learning pipelines. Furthermore, the authors demonstrated that existing bias can be reduced by choosing suitable transformers. Gohar *et al.* [29] carried out an empirical study to explain how fairness is affected in commonly used ensemble techniques. The results indicated that base learners contribute to bias in ensembles; however, this bias can be mitigated by using specific base learner configurations and selecting appropriate parameters. Additionally, previous research has highlighted the importance of aiding developers in reducing bias during the model training process. In contrast to these methods, which rely solely on empirical analysis, our work focuses on automatically explaining bias in a manner that is generalizable across various machine learning algorithms and fairness metrics.

Despite the progress in bias explanation and testing techniques, challenges remain in localizing fairness bugs and automatically explaining bias. In this dissertation, we aim to address the limitations of current bias explanation and testing methods by concentrating on automatically pinpointing sources of bias and providing more interpretable explanations for identified biases.

Bibliography

[1] 2024. Fairness Checker. https://github.com/tess022095/Fairness-Checker

[2] 2024. Fairness Contract. https://github.com/tess100766/FairnessContract

[3] Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna Wallach. 2018. A reductions approach to fair classification. In *International conference on machine learning*. PMLR, 60–69.

[4] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 625–635.

[5] Aws Albarghouthi and Samuel Vinitsky. 2019. Fairness-Aware Programming. In *Proceedings of the Conference on Fairness, Accountability, and Transparency* (Atlanta, GA, USA) *(FAT* '19)*. Association for Computing Machinery, New York, NY, USA, 211–219. https://doi.org/10.1145/3287560.3287588

[6] Solon Barocas, Moritz Hardt, and Arvind Narayanan. 2023. *Fairness and machine learning: Limitations and opportunities*. MIT press.

[7] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Probabilistic verification of fairness properties via concentration. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 118 (oct 2019), 27 pages. https://doi.org/10.1145/3360544

[8] Sumon Biswas and Hridesh Rajan. 2021. Fair preprocessing: towards understanding compositional fairness of data transformers in machine learning pipeline. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the*

Foundations of Software Engineering (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 981–993. https://doi.org/10.1145/3468264.3468536

[9] Sumon Biswas and Hridesh Rajan. 2023. Fairify: Fairness Verification of Neural Networks. In *ICSE'2023: The 45th International Conference on Software Engineering* (Melbourne, Australia).

[10] Miranda Bogen and Aaron Rieke. 2018. Help wanted: an examination of hiring algorithms, equity, and bias. https://api.semanticscholar.org/CorpusID:158203520

[11] Amanda Bower, Sarah N Kitchen, Laura Niss, Martin J Strauss, Alexander Vargas, and Suresh Venkatasubramanian. 2017. Fair pipelines. *arXiv preprint arXiv:1707.00391* (2017).

[12] Ajay Byanjankar, Markku Heikkilä, and Jozsef Mezei. 2015. Predicting credit risk in peer-to-peer lending: A neural network approach. In *2015 IEEE symposium series on computational intelligence*. IEEE, 719–725.

[13] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data mining and knowledge discovery* 21, 2 (2010), 277–292.

[14] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in machine learning software: Why? how? what to do?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 429–440.

[15] Zhenpeng Chen, Jie M. Zhang, Max Hort, Mark Harman, and Federica Sarro. 2024. Fairness Testing: A Comprehensive Survey and Analysis of Trends. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 137 (jun 2024), 59 pages. https://doi.org/10.1145/3652155

[16] Zhenpeng Chen, Jie M Zhang, Federica Sarro, and Mark Harman. 2022. A comprehensive empirical study of bias mitigation methods for software fairness. *arXiv preprint arXiv:2207.03277* (2022).

[17] Alexandra Chouldechova. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data* 5, 2 (2017), 153–163.

[18] Alexander D'Amour, Hansa Srinivasan, James Atwood, Pallavi Baljekar, David Sculley, and Yoni Halpern. 2020. Fairness is not static: deeper understanding of long term fairness via simulation studies. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. 525–534.

[19] Lucas Dixon, John Li, Jeffrey Sorensen, Nithum Thain, and Lucy Vasserman. 2018. Measuring and mitigating unintended bias in text classification. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*. 67–73.

[20] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*. 214–226.

[21] Cynthia Dwork and Christina Ilvento. 2018. Fairness under composition. *arXiv preprint arXiv:1806.06122* (2018).

[22] Michael Feffer, Martin Hirzel, Samuel C Hoffman, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2022. An Empirical Study of Modular Bias Mitigators and Ensembles. *arXiv preprint arXiv:2202.00751* (2022).

[23] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 259–268.

[24] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[25] Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. 2019. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the conference on fairness, accountability, and transparency*. 329–338.

[26] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. 498–510.

[27] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 498–510. https://doi.org/10.1145/3106237.3106277

[28] Giang Nguyen, Sumon Biwas, and Hridesh Rajan. 2023. *Replication Package of the ESEC/FSE 2023 Paper Entitled ”Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML*. https://doi.org/10.5281/zenodo.8280911

[29] Usman Gohar, Sumon Biswas, and Hridesh Rajan. 2023. Towards Understanding Fairness and its Composition in Ensemble Machine Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1533–1545. https://doi.org/10.1109/ICSE48619.2023.00133

[30] Nina Grgić-Hlača, Muhammad Bilal Zafar, Krishna P Gummadi, and Adrian Weller. 2018. Beyond distributive fairness in algorithmic decision making: Feature selection for procedurally fair learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[31] Moritz Hardt, Eric Price, and Nati Srebro. 2016. Equality of opportunity in supervised learning. *Advances in neural information processing systems* 29 (2016).

[32] Mitchell Hoffman, Lisa B Kahn, and Danielle Li. 2018. Discretion in hiring. *The Quarterly Journal of Economics* 133, 2 (2018), 765–800.

[33] Max Hort, Zhenpeng Chen, Jie M. Zhang, Mark Harman, and Federica Sarro. 2024. Bias Mitigation for Machine Learning Classifiers: A Comprehensive Survey. *ACM J. Responsib. Comput.* 1, 2, Article 11 (jun 2024), 52 pages. https://doi.org/10.1145/3631326

[34] Max Hort, Jie M Zhang, Federica Sarro, and Mark Harman. 2021. Fairea: A model behaviour mutation approach to benchmarking bias mitigation methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 994–1006.

[35] Philips George John, Deepak Vijaykeerthy, and Diptikalyan Saha. 2020. Verifying individual fairness in machine learning models. In *Conference on Uncertainty in Artificial Intelligence*. PMLR, 749–758.

[36] Faisal Kamiran and Toon Calders. 2012. Data preprocessing techniques for classification without discrimination. *Knowledge and information systems* 33, 1 (2012), 1–33.

[37] Lydia T Liu, Sarah Dean, Esther Rolf, Max Simchowitz, and Moritz Hardt. 2018. Delayed impact of fair machine learning. In *International Conference on Machine Learning*. PMLR, 3150–3158.

[38] Pranav Maneriker, Codi Burley, and Srinivasan Parthasarathy. 2023. Online Fairness Auditing through Iterative Refinement. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 1665–1676. https://doi.org/10.1145/3580305.3599454

[39] Giang Nguyen, Sumon Biswas, and Hridesh Rajan. 2023. Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/3611643.3616257

[40] Judea Pearl. 2009. Causal inference in statistics: An overview. (2009).

[41] Geoff Pleiss, Manish Raghavan, Felix Wu, Jon Kleinberg, and Kilian Q Weinberger. 2017. On fairness and calibration. *Advances in neural information processing systems* 30 (2017).

[42] Till Speicher, Hoda Heidari, Nina Grgic-Hlaca, Krishna P Gummadi, Adish Singla, Adrian Weller, and Muhammad Bilal Zafar. 2018. A unified approach to quantifying algorithmic unfairness: Measuring individual &group unfairness via inequality indices. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining.* 2239–2248.

[43] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, 401–416.

[44] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 98–108.

[45] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18).* Association for Computing Machinery, New York, NY, USA, 98–108. https://doi.org/10.1145/3238147.3238165

[46] Michael Veale, Max Van Kleek, and Reuben Binns. 2018. Fairness and accountability design needs for algorithmic support in high-stakes public sector decision-making. In *Proceedings of the 2018 chi conference on human factors in computing systems.* 1–14.

[47] Ke Yang, Biao Huang, Julia Stoyanovich, and Sebastian Schelter. 2020. Fairness-Aware Instrumentation of Preprocessing˜ Pipelines for Machine Learning. In *Workshop on Human-In-the-Loop Data Analytics (HILDA'20).*

[48] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rogriguez, and Krishna P Gummadi. 2017. Fairness constraints: Mechanisms for fair classification. In *Artificial Intelligence and Statistics*. PMLR, 962–970.

[49] Carlos Vladimiro González Zelaya. 2019. Towards explaining the effects of data preprocessing on machine learning. In *2019 IEEE 35th international conference on data engineering (ICDE)*. IEEE, 2086–2090.

[50] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. 2013. Learning fair representations. In *International conference on machine learning*. PMLR, 325–333.

[51] Brian Hu Zhang, Blake Lemoine, and Margaret Mitchell. 2018. Mitigating unwanted biases with adversarial learning. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*. 335–340.

[52] Hantian Zhang, Xu Chu, Abolfazl Asudeh, and Shamkant B. Navathe. 2021. OmniFair: A Declarative System for Model-Agnostic Group Fairness in Machine Learning. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2076–2088. https://doi.org/10.1145/3448016.3452787

[53] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 949–960.

CHAPTER 2. FIX FAIRNESS, DON’T RUIN ACCURACY: PERFORMANCE AWARE FAIRNESS REPAIR USING AUTOML

Giang Nguyen[1], Sumon Biwas[2], and Hridesh Rajan[3]

[1] Iowa State University, USA

[2] Carnegie Mellon University, USA

[3] Tulane University, USA

Modified from a manuscript published in *The proceedings of the 31st ACM Joint Meeting on European Software Engineering (ESEC/FSE ’23)*

2.1 Abstract

Machine learning (ML) is increasingly being used in critical decision-making software, but incidents have raised questions about the fairness of ML predictions. To address this issue, new tools and methods are needed to mitigate bias in ML-based software. Previous studies have proposed bias mitigation algorithms that only work in specific situations and often result in a loss of accuracy. Our proposed solution is a novel approach that utilizes automated machine learning (AutoML) techniques to mitigate bias. Our approach includes two key innovations: a novel optimization function and a fairness-aware search space. By improving the default optimization function of AutoML and incorporating fairness objectives, we are able to mitigate bias with little to no loss of accuracy. Additionally, we propose a fairness-aware search space pruning method for AutoML to reduce computational cost and repair time. Our approach, built on the state-of-the-art *Auto-Sklearn* tool, is designed to reduce bias in real-world scenarios. In order to demonstrate the effectiveness of our approach, we evaluated our approach on four fairness problems and 16 different ML models, and our results show a significant improvement over the baseline and existing bias

mitigation techniques. Our approach, *Fair-AutoML*, successfully repaired 60 out of 64 buggy cases, while existing bias mitigation techniques only repaired up to 44 out of 64 cases.

2.2 Introduction

Recent advancements in machine learning have led to remarkable success in solving complex decision-making problems such as job recommendations, hiring employees, social services, and education [11, 10, 39, 52, 38, 55, 21, 62, 22, 50, 24, 54, 2]. However, ML software can exhibit discrimination due to unfairness bugs in the models [3, 6]. These bugs can result in skewed decisions towards certain groups of people based on protected attributes such as race, age, or sex [30, 31].

To address this issue, the software engineering (SE) community has invested in developing testing and verification strategies to detect unfairness in software systems [8, 30, 1, 31, 61]. Additionally, the machine learning literature contains a wealth of research on defining different fairness criteria for ML models and mitigating bias [12, 20, 26, 37, 47, 49, 63, 64]. Various bias mitigation methods have been proposed to build fairer models. Some approaches mitigate data bias by adapting the training data [51, 16, 15]; some modify ML models during the training process to mitigate bias [57, 19, 40, 32, 59], and others aim to increase fairness by changing the outcome of predictions [1, 61, 65].

Despite these efforts, current bias mitigation techniques often come at the cost of decreased accuracy [41, 6]. Their effectiveness varies based on datasets, fairness metrics, or the choice of protected attributes [25, 26, 37, 18]. Hort et al. proposed Fairea [41], a novel approach to evaluate the effectiveness of bias mitigation techniques, which found that nearly half of the evaluated cases received poor effectiveness. Moreover, evaluations by Chen *et al.* also showed that in 25% of cases, bias mitigation methods reduced both ML performance and fairness [18].

Recent works [41, 59, 34] have shown that parameter tuning can successfully fix fairness bugs without sacrificing accuracy. By finding the best set of parameters, parameter tuning can minimize the error between the predicted values and the true values to reduce bias. This helps to ensure that the model is not overly simplified or too complex, which can lead to underfitting (high bias) or overfitting (low accuracy), respectively. By tuning the parameters, we can find the right balance between bias and accuracy, which leads to a model that generalizes well to different data or fairness metric. However, it is challenging to identify which parameter setting achieves the best fairness-accuracy trade-off [34].

Recent advancements in AutoML technology [29, 28, 42] have made it possible for both experts and non-experts to harness the power of machine learning. AutoML proves to be an effective option for discovering optimal parameter settings; however, currently there is a lack of focus on reducing bias within the AutoML techniques. Thus, we pose the following research questions: *Is it possible to utilize AutoML for the purpose of reducing bias? Is AutoML effective in mitigating bias? Does AutoML outperform existing bias reduction methods? Is AutoML more adaptable than existing bias mitigation techniques?*

We introduce *Fair-AutoML*, a novel technique that utilizes AutoML to fix fairness bugs in machine learning models. Unlike existing bias mitigation techniques, *Fair-AutoML* addresses their limitations by enabling efficient and fairness-aware Bayesian search to repair unfair models, making it effective for a wide range of datasets, models, and fairness metrics. The key idea behind *Fair-AutoML* is to use AutoML to explore as many configurations as possible in order to find the optimal fix for a buggy model. Particularly, *Fair-AutoML* enhances the potential of AutoML for fixing fairness bugs in two novel techniques: by generating a new optimization function that guides AutoML to fix fairness bugs without sacrificing accuracy, and by defining a new search space based on the specific input to accelerate the bug-fixing process. Together, these contributions enable *Fair-AutoML* to effectively fix fairness bugs across various datasets and fairness metrics. We have implemented *Fair-AutoML* on top of *Auto-Sklearn* [29], the state-of-the-art AutoML framework.

Fair-AutoML aims to effectively address the limitations of existing bias mitigation techniques by utilizing AutoML to efficiently repair unfair models across various datasets, models, and fairness metrics. We conduct an extensive evaluation of *Fair-AutoML* using 4 widely used datasets in the fairness literature [31, 61, 1] and 16 buggy models collected from a recent study [6]. The results demonstrate the effectiveness of our approach, as *Fair-AutoML* successfully repairs 60 out of 64 buggy cases, surpassing the performance of existing bias mitigation techniques which were only able to fix up to 44 out of 64 bugs in the same settings and training time.

Our main contributions are the following:

- We have proposed a novel approach to fix unfairness bugs and retain accuracy at the same time.
- We have proposed methods to generate the optimization function automatically based on an input to make AutoML fixing fairness bugs more efficiently.
- We have pruned the search space automatically based on an input to fix fairness bugs faster using AutoML.
- We have implemented our approach in a SOTA AutoML, *Auto-Sklearn* [29]. **The artifact is available here [33]**.

The paper is organized as follows: §**??** describes the background, §2.4 presents a motivation, §2.5 indicates the problem definition, §2.6 shows the *Fair-AutoML* approaches, §2.7 presents the our evaluation, §2.8 discusses the limitations and future directions of *Fair-AutoML*, §2.9 discusses the threats to validity of *Fair-AutoML*, §2.10 concludes.

2.3 Background

We begin by providing an overview of the background and related research in the field of software fairness.

2.3.1 Preliminaries

2.3.1.1 ML Software

Given an input dataset D split into a training dataset D_{train} and a validation dataset D_{val}, a ML software system can be abstractly viewing as mapping problem $M_{\lambda,c} : x \to y$ from inputs x to outputs y by learning from D_{train}. ML developers aims to search for a hyperparameter configuration $\lambda*$ and complementary components $c*$ for model M to obtain optimal fairness-accuracy on D_{val}. The complementary components can be ML algorithms combined with a classifier i.e., pre-processing algorithms.

2.3.1.2 AutoML

Given the search spaces Λ and C for hyperparameters and complementary components, AutoML aims to find $\lambda*$ and $c*$ to obtain the lowest value of the cost function (Equation 2.1):

$$M = \underset{\lambda \in \Lambda, c \in C}{\arg\min} Cost(M_{\lambda*,c*}, D_{val}) \tag{2.1}$$

$$(\lambda*, c*) = \underset{(\lambda,c)}{\arg\min} Loss(M_{\lambda,c}, D_{train}) \tag{2.2}$$

2.3.1.3 Fairness Measure.

We consider a problem where each individual in the population has a true label in $y = \{0, 1\}$. We assume a protected attribute $z = \{0, 1\}$, such as race, sex, and age, where one label is privileged (denoted 0) and the other is unprivileged (denoted 1). The predictions are $\hat{y} \in \{0, 1\}$ that need to be not only accurate with respect to y but also fair with respect to the protected attribute z.

Accuracy Measure. Accuracy is given by the ratio of the number of correct predictions by the total number of predictions.

$$\text{Accuracy} = (\#\text{ True positive} + \#\text{ True negative}) \ / \ \#\text{ Total}$$

Fairness Measure. We use four ways to define group fairness metrics, which are widely used in fairness literature [4, 30, 5]:

Disparate Impact (DI). DI is the proportion of the unprivileged group with the favorable label divided by the proportion of the privileged group with the favorable label [26, 63].

$$DI = \frac{Pr[\hat{y}=1|z=0]}{Pr[\hat{y}=1|z=1]}$$

Statistical Parity Difference (SPD). SPD quantifies the disparity between the favorable label's probability for the unprivileged group and the favorable label's probability for the privileged group [12].

$$SPD = Pr[\hat{y} = 1|z = 0] - Pr[\hat{y} = 1|z = 1]$$

Equal Opportunity Difference (EOD). EOD measures the disparity between the true-positive rate of the unprivileged group and the privileged group.

$$TPR_u = Pr[\hat{y} = 1|y = 1, z = 0]; TPR_p = Pr[\hat{y} = 1|y = 1, z = 1]$$

$$EOD = TPR_u - TPR_p$$

Average Absolute Odds Difference (AOD). AOD is the mean of the difference of true-positive rate and false-positive rate among the unprivileged group and privileged group [37].

$$FPR_u = P[\hat{y} = 1|y = 0, z = 0]; FPR_p = P[\hat{y} = 1|y = 0, z = 1]$$

$$AOD = \frac{1}{2} * |FPRu - FPRp| + |TPR_u - TPR_p|$$

To use all the metrics in the same setting, DI has been plotted in the absolute value of the log scale, and SPD, EOD, AOD have been plotted in absolute value [16, 41]. Thus, the bias score of a model is measured from 0, with lower scores indicating more fairness.

2.3.2 Related Work

2.3.2.1 Search Space Pruning

Search space pruning involves reducing the size or complexity of the search space in optimization or machine learning tasks. Pruning techniques are employed to accelerate the optimization process of AutoML by eliminating unpromising or redundant options, thus focusing computational resources on more promising areas of the search space. For example, Feurer *et al.* [29] introduce Auto-Sklearn 2.0, a novel approach aimed at enhancing the performance of Auto-Sklearn. This advancement involves constraining the search space to exclusively comprise iterative algorithms while eliminating feature preprocessing. This strategic adjustment streamlines the implementation of successive halving, as it reduces the complexity to a single fidelity type: the number of iterations. Otherwise, the incorporation of dataset subsets as an alternative fidelity would require additional consideration. Another innovative contribution comes from Cambronero *et al.*, who introduces AMS [13]. This method capitalizes on the wealth of source code repositories to streamline the search space for AutoML. Notably, AMS harnesses the power of unspecified complementary and functionally related API components. By leveraging these components, the search space for AutoML is pruned effectively. Diverging from prior research efforts, *Fair-AutoML* distinguishes itself by leveraging data characteristics to effectively trim down the search space. Notably, existing techniques in search space pruning primarily target accuracy enhancement within AutoML. In contrast, our innovative pruning methodology within *Fair-AutoML* is uniquely directed towards repairing unfair models.

2.3.2.2 AutoML Extension

AutoML aims to automate the process of building a high-performing ML model, but it has limitations. It can be costly and time-consuming to train, and it produces complex models that are difficult to understand. To address these limitations, software engineering researchers have developed methods to enhance AutoML performance, such as *AMS* [13] and *Manas* [53]. *AMS* utilizes source code repositories to create a new search space for AutoML, while *Manas* mines hand-developed models to find a better starting point for AutoML. The goal of these methods is to improve AutoML to maximize the accuracy. Different from these methods, *Fair-AutoML*, built on top of *Auto-Sklearn* [29], is the first to focus on repairing unfair models.

2.4 Motivation

The widespread use of machine learning in software development has brought attention to the issue of fairness in ML models. Although various bias mitigation techniques have been developed to address this issue, they have limitations. These techniques suffer from a poor balance between fairness and accuracy [41], and are not applicable to a wide range of datasets, metrics, and models [25, 26, 37]. To gain a deeper understanding of these limitations, we evaluate six different bias mitigation techniques using four fairness metrics, four datasets, and six model types. The evaluation criteria are borrowed from Fairea [41] and are presented in Table 2.1.

Fairea is designed to assess the trade-off between fairness and accuracy of bias mitigation techniques. The methodology of *Fairea* is demonstrated in Figure 2.1, where the fairness and accuracy of a bias mitigation technique on a dataset are displayed in a two-dimensional coordinate system. The baseline is established by connecting the fairness-accuracy points of the original model and the mitigation models on the dataset. *Fairea* evaluates the performance of the mitigation technique by altering the original model predictions and replacing a random subset of the predictions with other labels. The mutation degree ranges from 10% to 100% with a step-size of 10%. The baseline classifies the fairness-accuracy trade-off of a bias mitigation technique into five regions: lose-lose trade-off (**lose**), bad trade-off (**bad**), inverted trade-off (**inv**), good trade-off (**good**), and win-win trade-off (**win**). A technique reducing both accuracy and fairness would fall into the lose-lose trade-off region. If the trade-off is worse than the baseline, it would fall into the bad trade-off region. If the trade-off is better than the baseline, it would fall into the good trade-off region. If a bias mitigation method simultaneously decreases both bias and accuracy, it would fall into the inverted trade-off region. If the technique improves both accuracy and fairness, it would fall into the win-win trade-off region.

The results of the region classification of six bias mitigation techniques - *Reweighing* [47], *Disparate Impact Remover* [26], *Parfait-ML* [59], *Equalized Odds* [37], *FaX-AI* [35], *Reject Option Classification* [48] - are shown in Table 2.1. The evaluation was conducted on 64 buggy cases using different criteria such as fairness metrics and datasets. The case is identified as buggy when it falls below the *Fairea* baseline. The mean percentage of each technique falling into the corresponding regions is listed in each cell. The mean results provide a general overview of the current state of bias mitigation techniques. Further details on the performance of each individual bias mitigation technique can be found in Table 2.4 of our evaluation.

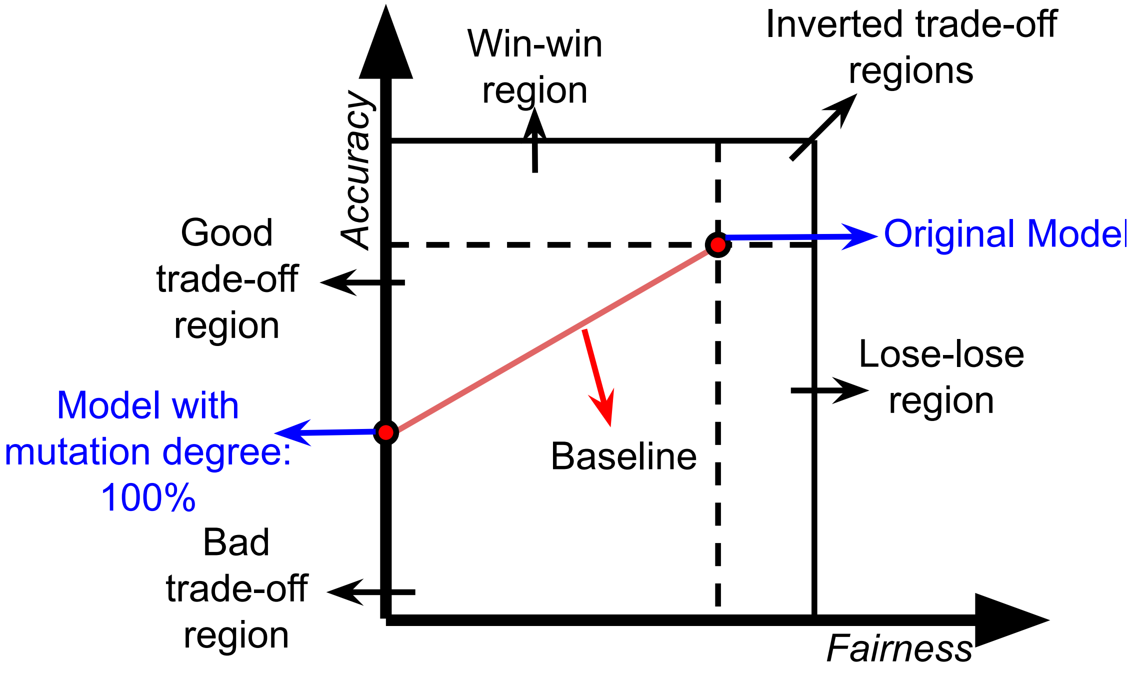


Figure 2.1: Baseline fairness-accuracy trade-off [41]

Table 2.1 illustrates that the majority of existing bias mitigation techniques have a poor fairness-accuracy trade-off across different datasets, fairness metrics, and classification models. Specifically, 39% of the cases show that these techniques perform worse than the original model, with 28% of the cases resulting in a poor trade-off and 11% resulting in a decrease in accuracy and an increase in bias. Additionally, Table 2.1 shows that the performance of these techniques varies depending on the input, as demonstrated by the different results obtained when using different datasets or painess metrics [25, 26, 37]. For example, the bias mitigation techniques had a high performance in 62% of the cases using the Adult dataset (55% for good trade-off region and 7% for win-win trade-off region), but only achieved 40% good effectiveness in the Bank dataset.

Hort *et al.* [41] have demonstrated that through proper parameter tuning, it is possible to address fairness issues in machine learning models without sacrificing accuracy. However, determining the optimal fairness-accuracy trade-off can be a challenge. Although AutoML can be effective in finding the best parameter settings, it does not specifically address bias reduction. This motivates the development of *Fair-AutoML*, a novel approach that utilizes Bayesian optimization to

Table 2.1: Mean proportions of mitigation cases that fall into each mitigation region

Criteria		Lose	Bad	Inv	Good	Win
Metric	DI	7%	31%	5%	43%	14%
	SPD	4%	36%	6%	40%	14%
	EOD	23%	15%	14%	40%	8%
	AOD	9%	30%	5%	40%	16%
Dataset	Adult [43]	18%	6%	14%	55%	7%
	Bank [44]	9%	44%	7%	23%	17%
	German [45]	6%	36%	2%	46%	10%
	Titanic [46]	11%	26%	3%	43%	17%
Mean		**11%**	**28%**	**7%**	**41%**	**13%**

Bad: bad trade-off region, Lose: lose-lose trade-off region, Inv: inverted trade-off region, Good: good trade-off region, Win: win-win trade-off region.

tune parameters and address fairness issues without hindering accuracy. *Fair-AutoML* is evaluated for its generality across different fairness metrics and datasets, and unlike other bias mitigation methods, it can be applied to any dataset or metric.

This work focuses on improving fairness quantitatively of buggy models instead of targeting a specific type of datasets and models. Our method is general since we utilize the power of AutoML to try as many configurations as possible to obtain the optimal fix; therefore, our method can work on various types of datasets and metrics. The rest of this work describes our approach, *Fair-AutoML*, that addresses the limitations of both existing bias mitigation methods and AutoML. As a demonstration, *Fair-AutoML* achieved good performance in 100% of the 16 buggy cases in the Adult dataset, while 75% of the mitigation cases showed a good fairness-accuracy trade-off, and the remaining 25% exhibited an improvement in accuracy without sacrificing bias reduction.

2.5 Problem Definition

This work aims to utilize AutoML to address issues of unfairness in ML software by finding a new set of configurations for the model that achieves optimal fairness-accuracy trade-off. Because fairness is an additional consideration beyond accuracy, the problem becomes a multi-objective optimization problem, requiring a new cost function that can optimize both fairness and accuracy simultaneously. To achieve this, we use a technique called weighted-sum scalarization (Equation 2.3) [23], which allows us to weigh the importance of different objectives and create a single scalar cost function.

$$A = \sum_{i=1}^{n} c_i * \beta_i \tag{2.3}$$

where, β_i denotes the relative weight of importance of c_i:

$$\sum_{i=1}^{n} \beta_i = 1 \tag{2.4}$$

In this work, we use a cost function (or objective function) that is a weighted-sum scalarization of two decision criteria: bias and accuracy. This cost function, as shown in Equation 2.5, assign weights to bias and accuracy in the cost function allow us to adjust the trade-off between the two criteria according to the specific problems:

$$Cost(M_{\lambda,c}, \mathcal{D}(z)) = \beta * f + (1 - \beta) * (1 - a) \tag{2.5}$$

We analyze the output of the buggy ML software (including bias and accuracy) to create a suitable cost function for each input. By analyzing the output, we are able to automatically estimate the weights of the cost function in order to balance fairness and accuracy for a specific problem. To the best of our knowledge, this is the first work that applies output analysis of the software to AutoML to repair unfair ML models.

However, using AutoML can be costly and time-consuming. To address this issue, we propose a novel method that automatically create new search spaces $\Lambda*$ and $C*$ based on different inputs to accelerate the bug-fixing process of AutoML. These new search spaces are smaller in size compared to the original ones, $|\Lambda * | < |\Lambda|$ and $|C * | < |C|$. Particularly, as shown in Equation 2.6, *Fair-AutoML* takes as input a ML model and a dataset with a protected attribute z, and aims to find $\lambda*$ and $c*$ in the smaller search space, in order to minimize the cost value.

$$M = \underset{\lambda* \in \Lambda*, c* \in C*}{\arg\min}\ Cost(M_{\lambda*,c*}, D_{val}(z)) \tag{2.6}$$

The technique of search space pruning in *Fair-AutoML* utilizes data characteristics to enhance bug-fixing efficiency. By shrinking the search spaces based on input analysis, *Fair-AutoML* can find

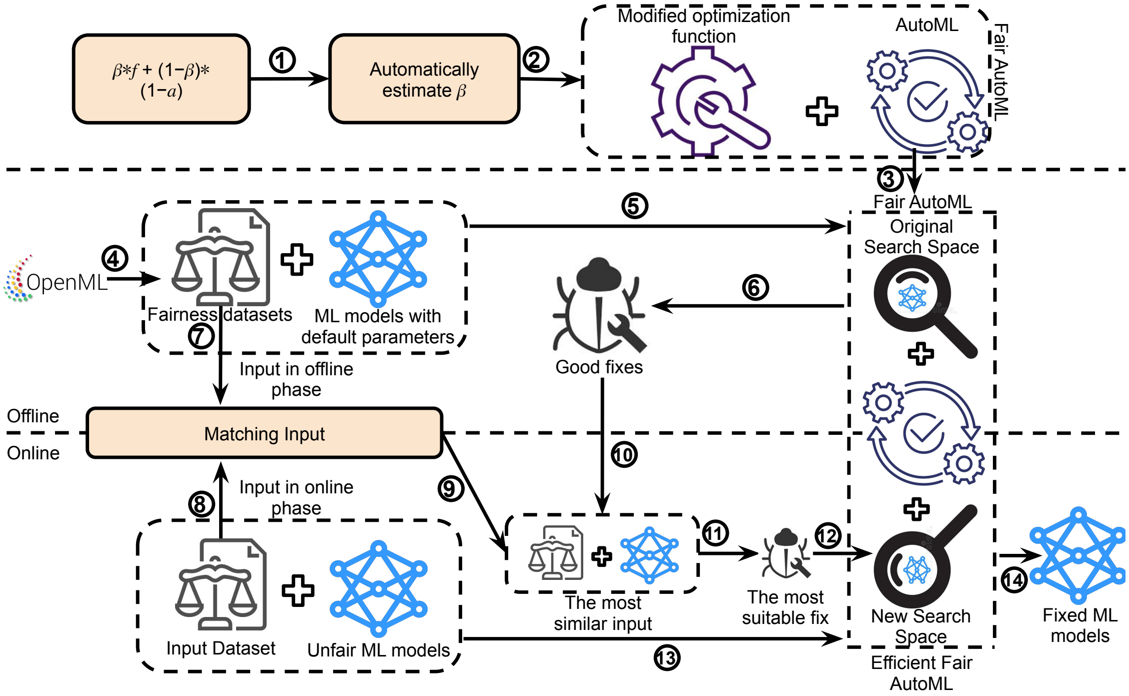


Figure 2.2: An Overview of *Fair-AutoML* Approach

better solutions more quickly. A set of predefined modifications to the ML model are pre-built and used as a new search space for new input datasets, reducing the time needed to fix buggy models. Our approach is based on previous works in AutoML [29], but updated and modified to tackle bias issues. To the best of our knowledge, we are the first to propose a search space pruning technique for fairness-aware AutoML.

2.6 Fair-AutoML

This section describes a detailed description of key components of *Fair-AutoML* (Figure 2.2): the dynamic optimization function (steps 1-3) and the search space pruning (steps 4-13).

2.6.1 Dynamic Optimization for Bias Elimination

We strive to eliminate bias in unfair models by utilizing Equation 2.5 as the objective function and determining the optimal value of β to minimize the cost function. In this section, we propose an approach to automatically estimate the optimal value of β for a specific dataset and a targeted model. This method ensures efficient correction of fairness issues while maintaining high predictive accuracy.

2.6.1.1 Upper bound of the Cost Function

To estimate the optimal value of β, the first step is to determine the upper bound of the cost function. This can be done by using a ”pseudo-model”, which is the 100% mutation degree model [41], as shown in the Figure 2.1. In other words, the pseudo-model always achieves the accuracy on any binary classification problem as follows:

$$a_0 = max(P(Y = 1), P(Y = 0)) \tag{2.7}$$

Given an input, the pseudo-model achieves an accuracy of a_0 and a bias value of f_0 on that input. We define the cost function, $Cost$, of the buggy ML model with accuracy a and bias value f on the input. As AutoML tries different hyperparameter configurations to fix the model, the values of a and f may change over time. The upper bound of the cost function is defined as Equations 2.8 and 2.9:

$$Cost(M_{\lambda,c}, \mathcal{D}(z)) < \beta * f_0 + (1 - \beta) * (1 - a_0) \tag{2.8}$$

$$\Leftrightarrow \beta * f + (1 - \beta) * (1 - a) < \beta * f_0 + (1 - \beta) * (1 - a_0) \tag{2.9}$$

The upper bound of the cost function is defined with the goal of repairing a buggy model so that its performance falls within a good/win-win trade-off region of fairness and accuracy. In other words, the accuracy of the repaired model must be higher than the accuracy of the pseudo-model. The repaired model must be better than the pseudo-model in terms of the cost function’s value. Since the pseudo-model has zero bias ($f_0 = 0$), the upper bound of the cost function is defined as follows (Equation 2.10):

$$\beta * f + (1 - \beta) * (1 - a) < (1 - \beta) * (1 - a_0) \tag{2.10}$$

2.6.1.2 Lower Bound of β

In this work, we desire to optimize the value of β in order to minimize bias as much as possible. The cost function used by *Fair-AutoML* is designed to balance accuracy and fairness, and increasing

β will place more emphasis on reducing bias. However, simply setting β to its highest possible value is not a viable option, as it may lead to low predictive accuracy and overfitting. We cannot accept models with poor predictive accuracy regardless of their low bias [56, 36]. To overcome this challenge, we aim to find the lower bound of β, which can be done based on the upper bound of the cost function. From Equation 2.10, we get:

$$\beta < \frac{a - a_0}{a - a_0 + f} \tag{2.11}$$

However, if the value of β is smaller than $\frac{a-a_0}{a-a_0+f}$, the optimization function $Cost$ will always meet its upper bound condition. If the value of β always satisfies the upper bound condition of the cost function regardless of accuracy and fairness, we can obtain a better optimization function by either increasing accuracy or decreasing bias. In this case, we cannot guide AutoML to produce a lower bias. Therefore, to guide AutoML produces an output with improved fairness, we set a lower bound for β as Equation 2.12:

$$\beta \geq \frac{a - a_0}{a - a_0 + f} \tag{2.12}$$

The intuition being that our method aims to increase the chance for AutoML to achieve better fairness. However, by setting $\beta < \frac{a-a_0}{a-a_0+f}$ and $a > a_0$ (we aim to find a model which has better accuracy than the pseudo-model), any value of bias (f) can satisfy upper bound condition of the cost function, which lower chance to obtain fairer models of AutoML. To increase this chance, we set $\beta \geq \frac{a-a_0}{a-a_0+f}$ and $a > a_0$. In this case, AutoML need to find better models that has lower bias to satisfy Equation 2.10. In other words, this lower bound condition indirectly forces bayesian optimization to search for lower bias models.

2.6.1.3 β Estimation

The final step is estimating the value of β based on its lower bound condition. Suppose that the buggy model achieves an accuracy of a_1 and a bias value of f_1 on that input. From the begining, we have: $a = a_1$ and $f = f_1$. In that time, the lower bound of β is $L = \frac{a_1-a_0}{a_1-a_0+f_1}$, so we have:

$$\beta = L + k, k \in [0, 1 - L] \tag{2.13}$$

Algorithm 1 Greedy Weight Identifier

```
1: **Input:** a dataset $D$ with protected attribute $z$, buggy model $M$ hyperparametered by $\lambda$, the increment value $\alpha$, the searching time $t$ and the threshold N
2: $\beta = \frac{a_1 - a_0}{a_1 - a_0 + f_1}$
3: $Cost(M_{\lambda,c}, \mathcal{D}(z)) = \beta * f + (1 - \beta) * (1 - a)$
4: $Cost_0(M_{\lambda,c}, \mathcal{D}(z)) = \ (1 - \beta) * (1 - a_0)$
5: count = 0
6: checker = False
7: **while** $t$ **do**
8:     $M_{\lambda *, c*} = \underset{\lambda \in \Lambda}{\arg\min} Cost(M_{\lambda,c}, \mathcal{D}(z))$
9:     count = count + 1
10:    **if** $Cost(M_{\lambda,c}, \mathcal{D}(z)) < Cost_0(M_{\lambda,c}, \mathcal{D}(z))$ **then**
11:        **if** checker = False **then**
12:            $\beta = \beta + \alpha$
13:            count = 0
14:    **if** $count \geq N$ and checker = False **then**
15:        $\beta = \beta$ - $\alpha$
16:        checker = True
17: **return** $M_\lambda *$
```

We present a greedy algorithm for estimating the value of β, which is detailed in Algorithm 1. Given a dataset D with a protected attribute z and a buggy model M (Line 1), we start by measuring the lower bound of β. Next, we run *Fair-AutoML* on the input under time constraint t with a value of β set to $\frac{a_1 - a_0}{a_1 - a_0 + f_1}$ (Line 2-8). As the algorithm searches, whenever *Fair-AutoML* finds a candidate model that meets the condition $Cost < Cost_0$ (Lines 10-12), the value of β is slightly increased by α (Line 10-12). If after N tries, *Fair-AutoML* cannot find a model that satisfies the condition, the final value of β is set to $\beta = \beta$ - α for the remaining search time to prevent overfitting from an excessively high value of β (Lines 13-15). The algorithm returns the best model found (Line 16).

2.6.2 Search Space Pruning for Efficient Bias Elimination

We propose a solution to speed up the Bayesian optimization process in *Fair-AutoML* by implementing search space pruning. This technique takes advantage of data characteristics to automatically reduce the size of the search space in AutoML, thus improving its efficiency. Our approach includes two phases: the offline phase and the online phase. The offline phase trains a set

Algorithm 2 Database Building

```
Input: a dataset $D$ with protected attribute $z$, a model $M$ with default hyperparameters $\lambda$. Running time t.
d = ∅
dev = 1
database = {}
space = {}
count = 0
while count ≤ n do
    count = count + 1
    while t do
        $M_\lambda * = \arg\min Cost(M_\lambda, \mathcal{D}(z))$
        $d = d \cup M_\lambda *$
    kBestPipelines = top_k(d)
    mBestComponents = top_m(kBestPipelines)
    for model ∈ kBestPipelines do
        for para ∈ model do
            space[para] = space[para]∪[para.val]
for para ∈ space do
    if para is numerical then
        no_outliers = ∅
        for i ∈ space[para] do
            if $|i - \overline{space[para]}| < dev * \sigma(space[para])$ then
                no_outliers = no_outliers ∪ space[para][i]
    space[para] = [min(no_outliers), max(no_outliers)]
database[input] = (space, mBestComponents)
return database
```

of inputs multiple times to gather a collection of hyperparameters and complementary components for each input, forming a pre-built search space. In the online phase, when a new input is encountered, it is matched against the inputs stored in our database to find a matching pre-built search space, which is then utilized to repair the buggy model. This approach effectively replaces the original search space of *Fair-AutoML*, making the Bayesian optimization process much faster. Search space pruning has already been successfully applied before [13, 28]; however, this is the first application of data characteristics to prune the search space for fairness-aware AutoML.

2.6.2.1 Offline Phase

This phase constructs a set of search spaces for *Fair-AutoML* based on different inputs. It is important to note that the input format in the offline phase must match that of the online phase, which includes a dataset with a protected attribute and a ML model. This ensures that the pre-built search spaces created in the offline phase can be effectively utilized in the online phase.

Input In the offline phase, we collect a set of inputs to build search spaces for *Fair-AutoML*. The inputs are obtained as follows. Firstly, we mine machine learning datasets from *OpenML*, considering only the 3425 active datasets that have been verified to work properly. Secondly, to ensure that the mined datasets are relevant to the fairness problem, we only collect datasets that contain at least one of the following attributes: *age*, *sex*, *race* [17]. In total, we collected 231 fairness datasets. Thirdly, for each mined dataset, we use all available protected attributes. For example, when dealing with datasets that contain multiple protected attributes, such as the *Adult* dataset that includes *sex* and *race* as protected attributes, we treat them as distinct inputs for the dataset. Finally, we use the default values for the hyperparameters of the input ML model in the offline phase, as we do not know the specific values that will be used in the online phase.

Database building To build a pre-defined search space database, we use the algorithm outlined in Algorithm 2 to obtain a pre-built search space for each collected input in order to fix the buggy model. This process involves training a fairness dataset with a specific protected attribute and ML model multiple times using *Fair-AutoML*, collecting the top k best pipelines found, and extracting parameters from these pipelines. In particular, we use *Fair-AutoML* to train the fairness dataset with a specific protected attribute and a ML model for n iterations (Line 7-11). We then gather the top k best pipelines, including a classifier and complementary components, found by *Fair-AutoML* according to the optimization function's value (Line 12). This results in $k * n$ total pipelines. From these pipelines, we extract and store the m most frequently used complementary components in the database (Line 13). For each classifier parameter, we also store its value (Lines 14-16). This results in k∗n values being stored for each hyperparameter. If a

Algorithm 3 Input Matching

```
Input: a input dataset $D$ with the protected attribute $z$, the number of data points $p$, the number of features $f$, lower bound $L$, a buggy model $M$, and a database.
dist = {}
for $d_i$ in database do
    dist[$d_i$] = $|f_i - f| + |p_i - p|$
similarDataset = min(dist, key=dist.get)
dist = {}
for $z_i$ in similarDataset do
    dist[$d_i$] = $|L_i - L|$
similarAttribute = min(dist, key=dist.get)
similarModel = M with default parameter
return similarDataset, similarAttribute, similarModel
```

hyperparameter is categorical and its values are sampled from a set of different values, we store all its unique values in the database. If a hyperparameter is numerical and its values are sampled from a uniform distribution, we remove any outliers and store the range of values from the minimum to the maximum in the database (Lines 17-23). After this process, we have collected the pre-built search space for the input (Lines 24-25). We believe that two similar inputs may have similar buggy models and fixes, so the pre-built search space is built based on the best models found by *Fair-AutoML* from similar inputs, making it a reliable solution for fixing buggy models.

2.6.2.2 Online Phase

This phase utilizes a pre-built search space from the database to fix a buggy model for a given dataset by replacing the original search space with the pre-built one.

Search space pruning Our approach of search space pruning in *Fair-AutoML* improves the bug fixing performance by reducing the size of the hyperparameter tuning space. Algorithm 3 is used to match the input dataset, protected attribute, and ML model to the most similar input in the database. Firstly, data characteristics such as the number of data points and features are used to match the new dataset with the most similar one in the database [28]. L1 distance is computed between the new dataset and each mined dataset in the space of data characteristics to determine the closest match. We consider that the most similar dataset to the new dataset is the nearest one

(Line 2-5). Secondly, we compute the lower bound $L = \frac{a_1 - a_0}{a_1 - a_0 + f_1}$ of β of the new input. We then estimate the lower bound of β of all the protected attributes of the matched dataset and select the attribute whose lower bound is closest to L (Line 6-9). Lastly, two similar inputs must use the same ML algorithm (Line 10). The matching process is carried out in the order of dataset matching, protected attribute matching, and ML algorithm matching. The pre-built search space of the similar input is then used as the new search space for the new input.

2.7 Evaluation

In this section, we describe the design of the experiments to evaluate the efficient of *Fair-AutoML*. We first pose research questions and discuss the experimental details. Then, we answer research questions regarding the efficiency and adaptability of *Fair-AutoML*.

RQ1: Is *Fair-AutoML* effective in fixing fairness bugs? To answer this question, we quantify the number of fairness bugs that *Fair-AutoML* is able to repair compared to existing methods, allowing us to assess the capability of an AutoML system in fixing fairness issues.

RQ2: Is *Fair-AutoML* more adaptable than existing bias mitigation techniques? The adaptability of a bias mitigation technique indicates its performance across a diverse range of datasets/metrics. So, we analyze the effectiveness of *Fair-AutoML* and existing bias mitigation techniques on different dataset/metrics to assess the adaptability of an AutoML system on fix fairness bugs.

RQ3: Are dynamic optimization function and search space pruning effective in fixing fairness bugs? To answer this question, we assess the performance of *Auto-Sklearn*, both with and without the dynamic optimization function and search space pruning, to demonstrate the impact of each proposed approach.

2.7.1 Experiment

2.7.1.1 Benchmarks

We evaluated our method using real-world fairness bugs sourced from a recent empirical study [6], with our benchmark consisting of 16 models collected from Kaggle covering five distinct types: XGBoost (**XGB**), Random Forest (**RF**), Logistic Regression (**LRG**), Gradient Boosting (**GBC**), Support Vector Machine (**SVC**). We use four popular datasets for our evaluation [10, 60, 61]:

The **Adult Census** (race) [43] comprised of 32,561 observations and 12 features that capture the financial information of individuals from the 1994 U.S. census. The objective is to predict whether an individual earns an annual income greater than 50K.

The **Bank Marketing** (age) [44] has 41,188 data points with 20 features including information on direct marketing campaigns of a Portuguese banking institution. The classification task aims to identify whether the client will subscribe to a term deposit.

The **German Credit** (sex) [45] has 1000 observations with 21 features containing credit information to predict good or bad credit.

The **Titanic** (sex) [46] has 891 data points with 10 features containing individual information of Titanic passengers. The dataset is used to predict who survived the Titanic shipwreck.

2.7.1.2 Evaluated Learning Techniques

We examined the performance of *Fair-AutoML* and other supervised learning methods addressing discrimination in binary classification including all three types of bias mitigation techniques and Auto-ML techniques.

Bias mitigation methods We investigate all three types of bias mitigation methods: pre-processing, in-processing, post-processing. We select widely-studied bias mitigation methods for each category:

- The **pre-processing** includes *Reweighing* (**R**) [47], *Disparate Impact Remover* (**DIR**) [26].

- The **in-processing** includes *Parfait-ML* (**PML**) [59].
- The **post-processing** includes *Equalized Odds* (**EO**) [37], *FaX-AI* (**FAX**) [35], *Reject Option Classification* (**ROC**) [48].

Auto-Sklearn We explore the efficiency of *Auto-Sklearn* (**AS**) [29] on mitigating bias in unfair model. Although, *Auto-Sklearn* does not seek to decrease bias, we compare its performance with *Fair-AutoML* to demonstrate the efficient of our techniques in guiding Auto-ML to repair fairness bugs.

Fair-AutoML We create 4 versions of *Fair-AutoML* in this evaluation representing for *Fair-AutoML* with different cost functions:

- **T1** uses $\beta * DI + (1 - \beta) * (1 - accuracy)$ as a cost function.
- **T2** uses $\beta * SPD + (1 - \beta) * (1 - accuracy)$ as a cost function.
- **T3** uses $\beta * EOD + (1 - \beta) * (1 - accuracy)$ as a cost function.
- **T4** uses $\beta * AOD + (1 - \beta) * (1 - accuracy)$ as a cost function.

2.7.1.3 Experimental Configuration

Experiments were conducted using Python 3.6 on Intel Skylake 6140 processors. *Fair-AutoML* leverages the capabilities of *Auto-Sklearn* [29], taking advantage of its automatic optimization of the best ML model for a given dataset. We tailored *Auto-Sklearn* to better fit our method in two ways: (1) its search space was restricted to the type of the faulty classifier - for example, if the faulty classifier is Random Forest, *Auto-Sklearn* will only optimize the hyperparameters and identify complementary components for that specific classifier. (2) The faulty model was set as the default model for *Auto-Sklearn*. These modifications are features of *Auto-Sklearn* that we utilized.

Methodology Configuration We selected an increment value of α for β of 0.05 to balance the time between β search and model fixing processes. The user can opt for a more accurate value of β by decreasing the increment value and using a longer search time. To conduct search space pruning, we ran *Fair-AutoML* 10 times (n) with a 1-hour search time (t) to gather the best ML pipelines [9]. From each run, we collected the top 10 pipelines (k), resulting in 100 models per input. This pre-built search space includes a set of hyperparameters and the top 3 most frequently used complementary components (m). We have explored other parameter settings, but these have proven to provide optimal results.

Evaluation Configuration We evaluate each tool on each buggy scenario 10 times using a random re-split of the data based on a 7:3 train-test split ratio [41]. The runtime for each run of *Fair-AutoML* and *Auto-Sklearn* is approximately one hour [29, 28]. The mean performance of each method is calculated as the average of the 10 runs, which is a commonly used practice in the fairness literature [4, 6, 14]. Our evaluation targets fixing 16 buggy models for 4 fairness metrics, resulting in a total of 64 buggy cases.

2.7.2 Effectiveness (RQ1)

We evaluate the effectiveness of *Fair-AutoML* by comparing it with *Auto-Sklearn* and existing bias mitigation techniques based on *Fairea* baseline. The comparisons are based on the following rules:

- **Rule 1**: A model is considered successfully repaired when its post-mitigation mean accuracy and fairness falls into win-win/good trade-off regions.
- **Rule 2**: A model that falls in the win-win region is always better than one falling into any other region.
- **Rule 3**: If two models are in the same trade-off region, the one with lower bias is preferred.

Our comparison rules for bug-fixing performance were established based on Fairea and our evaluations. Firstly, we define a successful bug fix as a fixed model that falls within the win-win or

Table 2.2: Trade-off assessment results of *Fair-AutoML*, *Auto-Sklearn*, and mitigation techniques

	M	Metric	T1	T2	T3	T4	AS	R	DIR	PML	EO	FAX	ROC	M	Metric	T1	T2	T3	T4	AS	R	DIR	PML	EO	FAX	ROC
Adult Census	RF	Acc	0.010	0.001	0.005	-0.003	0.016	0.009	0.004	0.008	-0.006	0.002	-0.047	LRG	Acc	-0.013	-0.022	-0.009	-0.032	-0.001	-0.056	-0.055	-0.054	-0.057	-0.057	-0.061
		DI	**0.096**	**0.011**	**0.118**	0.095	**0.058**	**0.337**	inv	inv	0.292	**0.000**	0.445		DI	0.190	0.410	0.212	0.179	0.040	0.569	0.326	lose	0.375	bad	0.318
		SPD	**0.023**	**0.024**	**0.038**	0.048	**0.016**	**0.055**	inv	inv	0.047	inv	0.055		SPD	0.054	0.075	0.029	0.059	0.005	0.097	0.079	bad	0.083	0.076	0.072
		EOD	**0.019**	inv	**0.014**	0.020	**0.008**	inv	inv	inv	0.041	inv	lose		EOD	0.048	0.020	0.044	0.049	lose	0.057	0.056	0.060	0.067	0.052	0.059
		AOD	**0.028**	inv	0.030	0.035	**0.021**	**0.005**	**0.001**	inv	0.051	**0.007**	bad		AOD	0.085	0.075	0.079	0.089	0.039	0.096	0.094	0.091	0.103	0.094	0.095
	XGB	Acc	-0.019	-0.052	-0.017	-0.015	-0.001	-0.001	-0.018	-0.005	-0.035	0.001	-0.056	GBC	Acc	-0.023	-0.046	-0.014	-0.034	0.002	-0.006	-0.011	-0.034	-0.018	-0.001	-0.059
		DI	0.183	0.148	0.143	0.156	0.004	0.378	lose	lose	0.330	inv	0.456		DI	0.124	0.188	0.104	0.147	inv	0.387	0.027	lose	0.227	0.076	0.438
		SPD	0.028	0.074	0.023	0.030	0.003	0.058	lose	lose	0.051	inv	0.054		SPD	bad	0.055	0.012	0.029	inv	0.058	bad	lose	0.035	0.011	0.047
		EOD	0.036	0.041	0.037	0.030	lose	lose	lose	0.003	0.044	inv	lose		EOD	bad	0.025	0.013	0.024	inv	lose	lose	lose	0.037	0.010	lose
		AOD	0.055	0.064	0.047	0.053	0.017	0.009	lose	0.010	0.066	**0.020**	bad		AOD	0.037	0.055	0.041	0.050	**0.018**	0.031	0.031	lose	0.063	0.041	bad
Bank Marketing	RF	Acc	-0.012	-0.023	-0.001	-0.007	0.000	-0.008	-0.014	-0.008	-0.034	0.001	-0.082	XGB2	Acc	-0.001	-0.008	0.000	0.002	0.000	0.001	-0.080	-0.005	-0.075	0.000	-0.143
		DI	0.103	0.224	0.031	0.038	0.000	0.210	lose	bad	bad	**0.129**	bad		DI	0.158	0.236	**0.166**	**0.097**	lose	**0.312**	bad	bad	bad	**0.016**	bad
		SPD	0.035	bad	0.007	0.027	lose	0.065	bad	0.031	bad	**0.021**	bad		SPD	0.032	0.051	**0.028**	**0.021**	lose	**0.053**	bad	bad	bad	**0.002**	bad
		EOD	lose	bad	lose	lose	lose	lose	lose	0.029	bad	**0.001**	bad		EOD	lose	lose	**0.003**	**0.012**	lose	inv	inv	0.054	bad	inv	inv
		AOD	0.033	0.039	0.032	0.032	0.016	bad	0.020	0.046	bad	**0.031**	bad		AOD	0.023	0.018	**0.026**	**0.027**	lose	inv	inv	0.040	bad	**0.014**	bad
	XGB1	Acc	-0.002	0.002	0.007	0.000	-0.001	-0.007	-0.019	-0.005	-0.058	0.003	-0.018	GBC	Acc	-0.003	-0.007	0.014	0.011	-0.002	0.000	-0.066	-0.034	-0.058	0.001	-0.104
		DI	0.098	**0.348**	**0.114**	**0.092**	lose	0.222	0.174	bad	bad	**0.067**	0.183		DI	0.010	0.069	**0.011**	**0.014**	lose	**0.332**	bad	bad	bad	inv	bad
		SPD	0.023	**0.062**	**0.023**	**0.020**	lose	0.042	bad	0.029	bad	**0.013**	bad		SPD	lose	0.028	inv	inv	lose	**0.052**	bad	bad	bad	inv	bad
		EOD	0.021	inv	**0.011**	**0.018**	lose	lose	lose	0.064	bad	inv	lose		EOD	lose	lose	inv	inv	lose	inv	lose	bad	bad	inv	inv
		AOD	0.043	**0.040**	**0.043**	**0.046**	lose	lose	0.038	0.054	bad	**0.020**	0.043		AOD	0.014	0.023	**0.027**	**0.025**	lose	**0.017**	lose	bad	bad	**0.012**	bad
German Credit	RF	Acc	-0.020	-0.027	-0.012	-0.012	-0.011	-0.016	-0.007	-0.016	-0.529	-0.004	-0.446	SVC	Acc	-0.011	-0.022	-0.019	-0.015	-0.007	-0.013	-0.009	-0.044	-0.042	-0.012	-0.203
		DI	bad	0.076	lose	0.060	lose	0.066	0.039	0.076	0.095	bad	bad		DI	0.109	0.130	0.103	0.108	0.035	bad	bad	bad	bad	0.111	bad
		SPD	bad	0.052	lose	0.039	lose	0.044	0.025	0.052	0.068	bad	bad		SPD	0.077	0.092	0.070	0.078	0.021	bad	bad	bad	bad	0.078	bad
		EOD	0.044	0.062	0.045	0.059	0.033	0.054	0.042	0.079	0.064	0.023	bad		EOD	0.101	0.101	0.101	0.081	0.068	0.039	0.039	bad	0.107	0.092	bad
		AOD	lose	bad	lose	lose	lose	lose	lose	0.015	0.044	lose	bad		AOD	0.019	0.034	bad	0.027	lose	lose	lose	bad	0.059	0.016	bad
	XGB	Acc	0.003	-0.009	-0.014	-0.017	-0.016	-0.028	-0.026	0.005	-0.043	-0.006	-0.443	KNN	Acc	0.001	0.000	0.010	-0.005	0.000	0.005	0.009	-0.010	-0.002	-0.049	-0.420
		DI	**0.070**	0.091	0.119	0.101	bad	bad	bad	**0.051**	bad	0.035	lose		DI	**0.112**	**0.104**	**0.065**	0.123	0.038	**0.071**	**0.046**	0.160	0.150	0.135	bad
		SPD	**0.050**	0.065	0.082	0.069	bad	0.060	0.048	**0.038**	bad	0.025	bad		SPD	**0.072**	**0.075**	**0.043**	0.090	0.027	**0.047**	**0.028**	0.120	0.111	0.098	bad
		EOD	**0.069**	0.074	0.073	0.083	0.036	0.064	0.064	**0.103**	0.091	0.055	bad		EOD	**0.104**	**0.080**	**0.085**	0.076	0.066	**0.066**	**0.066**	0.128	0.115	0.002	bad
		AOD	**0.020**	0.020	0.041	0.037	lose	bad	bad	**0.015**	0.064	bad	bad		AOD	**0.017**	**0.011**	inv	0.034	lose	inv	inv	0.072	0.065	0.035	bad
Titanic	RF	Acc	-0.098	-0.130	-0.129	-0.128	-0.014	-0.166	-0.021	-0.010	-0.179	0.005	-0.178	GBC	Acc	-0.035	-0.076	-0.136	-0.126	0.065	-0.139	-0.015	-0.048	-0.189	-0.023	-0.165
		DI	1.549	1.864	1.848	1.849	lose	0.536	0.160	lose	2.024	**0.449**	2.303		DI	0.501	0.885	1.411	1.303	**0.092**	0.385	0.038	bad	1.769	bad	1.991
		SPD	0.395	0.571	0.551	0.545	lose	bad	bad	lose	bad	**0.153**	0.651		SPD	0.121	0.275	0.462	0.447	inv	bad	bad	0.285	bad	0.187	0.641
		EOD	0.274	0.404	0.445	0.446	lose	lose	lose	lose	0.481	**0.045**	bad		EOD	bad	bad	bad	bad	**0.058**	lose	lose	0.280	bad	0.116	0.426
		AOD	0.477	0.556	0.534	0.601	0.062	bad	0.133	0.097	0.618	**0.336**	bad		AOD	0.183	0.305	0.467	0.445	**0.176**	bad	0.081	0.374	0.568	0.306	bad
	LRG	Acc	-0.021	-0.084	-0.091	0.000	-0.007	-0.159	0.015	0.009	-0.227	-0.023	-0.152	XGB	Acc	-0.079	-0.101	-0.099	-0.110	-0.019	-0.129	-0.006	0.008	-0.157	0.009	-0.159
		DI	0.743	1.619	1.673	**0.149**	0.086	0.597	**0.214**	**0.643**	bad	bad	2.557		DI	1.364	1.701	1.470	1.663	lose	0.671	0.203	**0.539**	1.811	Inv	2.172
		SPD	0.101	0.552	0.597	**0.113**	0.063	bad	**0.007**	**0.115**	bad	0.312	0.785		SPD	0.280	0.542	0.406	0.491	lose	bad	0.065	**0.051**	0.567	Inv	0.642
		EOD	bad	0.467	0.557	**0.021**	lose	lose	**0.009**	**0.171**	bad	0.275	0.623		EOD	bad	0.400	0.285	0.389	lose	lose	lose	**0.058**	0.473	Inv	0.423
		AOD	0.140	0.562	0.632	**0.179**	0.101	bad	**0.067**	**0.214**	bad	0.420	bad		AOD	0.300	0.532	0.356	0.524	lose	bad	0.174	**0.181**	0.585	**0.062**	bad

Each cell shows the accuracy/bias difference between the original and repaired models. Accuracy difference is calculated as new accuracy - old accuracy, while bias difference is old bias - new bias. A positive value indicates improvement. Bias differences are provided for methods in the good or win-win regions (regular or bold numbers), while other regions are labeled accordingly. The most effective bug-fixing method is highlighted in blue.

Table 2.3: Fair-AutoML (FA) vs bias mitigation methods in fixing fairness bugs

	FA	AS	R	DIR	PML	EO	FAX	ROC
# bugs fixed	60	28	35	30	36	37	44	23
# best models	19	4	9	0	10	9	9	3

The results in this table are derived from the data presented in Table 2.2. The row *# bugs fixed* indicates the number of cases where the technique falls into either the win-win or good trade-off region. The row *# best models* represents the number of instances where a bias mitigation technique outperforms all other methods.

good trade-off regions, as these regions demonstrate improved fairness-accuracy trade-offs compared to the baseline in Fairea. Secondly, when comparing successfully fixed models in different trade-off regions (win-win versus good), we consider the win-win models to be superior as they offer improved fairness and accuracy. Lastly, for models that fall within the same trade-off region, the one with lower bias is deemed to be better, as our goal is to fix unfair models. Our evaluations then consider two aspects of the bug-fixing performance: the number of successful bug fixes and the number of times a bias mitigation method outperforms others.

2.7.2.1 Is *Fair-AutoML* Effective in Fixing Fairness Bugs?

The results presented in Table 2.3 show that *Fair-AutoML* was effective in resolving 60 out of 64 (94%) fairness bugs, while *Auto-Sklearn* only fixed 28 out of 64 (44%) and bias mitigation techniques resolved up to 44 out of 64 (69%). This indicates that *Auto-Sklearn* alone was not effective in reducing bias, however, our methods were successful in enhancing AutoML to repair fairness bugs. Moreover, *Fair-AutoML* was able to repair more cases than other bias mitigation techniques, which often resulted in lower accuracy for lower bias. This highlights the effectiveness of our approaches in guiding AutoML towards repairing models for better trade-off between fairness and accuracy compared to the Fairea baseline.

2.7.2.2 Does *Fair-AutoML* Outperform Bias Reduction Techniques?

Fair-AutoML demonstrated superior performance in fixing fairness bugs compared to other bias mitigation techniques. The results presented in Table 2.3 indicate that 63 out of 64 buggy cases were fixed by *Fair-AutoML*, *Auto-Sklearn*, or bias mitigation techniques. Among the repaired buggy cases, *Fair-AutoML* outperformed other techniques 19 times (30%). On the other hand, *Auto-Sklearn* outperformed *Fair-AutoML* and bias mitigation techniques only 4 times (6%), and bias mitigation techniques outperformed other techniques 10 times at most (16%). This highlights that *Fair-AutoML* is often more effective in improving fairness and accuracy simultaneously or reducing more bias than other bias mitigation techniques.

2.7.3 Adaptability (RQ2)

To assess the adaptability of *Fair-AutoML*, we measure the proportions of each evaluated tools that fall into each fairness-accuracy trade-off region in different categories: fairness metric and dataset (Table 2.4). To further evaluate the adaptability of *Fair-AutoML*, instead of using our prepared models and datasets, we used the benchmark [58] of Parfait-ML to evaluate *Fair-AutoML*. Particularly, we evaluate *Fair-AutoML* and Parfait-ML on three different ML models (Decision Tree, Logistic Regression, Random Forest) on two datasets (Adult Census and COMPAS) (Table 2.5 and Figure 2.3).

2.7.3.1 Is *Fair-AutoML* More Adaptable Than Existing Bias Mitigation Techniques and *Auto-Sklearn*?

Table 2.4 shows *Fair-AutoML* demonstrates exceptional repair capabilities across various datasets and fairness metrics, with a high rate of success in fixing buggy models. For example, in the Adult Census, Bank Marketing, German Credit, and Titanic datasets, *Fair-AutoML* (T4) repaired 100%, 82%, 94%, and 94% of the models, respectively. Similarly, in the DI, SPD, EOD, and AOD fairness metrics, *Fair-AutoML* (T4) achieved repair rates of 100%, 94%, 82%, and 94%. On the other hand, bias mitigation methods often show inconsistent results. For instance, Equalized Odds repaired all buggy cases in *Adult Census* but none in *Bank Marketing.* In fact, our methods effectively guides AutoML in hyperparameter tuning to reduce bias, leading to superior repair performance across different datasets and metrics.

Table 2.4: Proportion of *Fair-AutoML*, *Auto-Sklearn*, and mitigation techniques that fall into each mitigation region

Method	Fairness Metric																				Dataset																			
	DI					SPD					EOD					AOD					Adult Census					Bank Marketing					German Credit					Titanic				
	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win	Lose	Bad	Inv	Good	Win
T1	0	6	0	75	19	6	12	0	63	19	18	25	0	38	19	6	0	0	75	19	0	12	0	63	25	25	0	0	75	0	6	13	0	31	50	0	19	0	81	0
T2	0	0	0	81	19	0	0	0	81	19	12	13	13	56	6	0	6	6	75	13	0	0	12	75	13	12	13	6	50	19	0	6	0	69	25	0	6	0	94	0
T3	6	0	0	63	31	6	0	6	63	25	6	6	6	56	26	6	6	6	56	26	0	0	0	75	25	0	12	19	13	56	19	6	6	50	19	0	6	0	94	0
T4	0	0	0	75	25	0	0	6	75	19	6	6	6	63	19	6	0	0	69	25	0	0	0	100	0	5	0	13	19	63	6	0	0	94	0	0	6	0	94	0
Avg	2	2	0	73	23	3	3	3	70	21	10	13	6	53	18	5	3	3	69	20	0	3	3	78	16	11	6	9	39	35	8	6	2	61	23	0	9	0	91	0
AS	38	6	6	38	12	44	6	13	31	6	56	0	6	25	13	50	0	0	31	19	13	0	19	44	24	88	0	0	12	0	38	12	0	50	0	50	0	6	25	19
R	0	13	0	62	25	0	31	0	44	25	50	0	19	25	6	18	38	13	19	12	13	0	6	62	19	19	6	19	25	31	13	25	6	38	18	25	50	0	25	0
DIR	13	25	6	44	12	6	50	6	25	13	50	0	13	25	12	25	6	13	44	12	31	6	19	38	6	31	38	13	18	0	13	25	6	38	18	19	13	0	43	25
PML	25	38	6	12	19	19	25	6	31	19	13	13	6	49	19	6	13	6	56	19	44	6	25	25	0	0	50	0	50	0	0	25	0	50	25	19	6	0	25	50
EO	0	44	0	56	0	0	56	0	44	0	0	38	0	62	0	0	31	0	69	0	0	0	0	100	0	0	100	0	0	0	0	25	0	75	0	0	44	0	56	0
FAX	0	25	19	25	31	0	6	25	44	25	0	0	38	50	12	6	6	0	38	50	0	6	31	44	19	0	0	0	31	69	6	19	0	75	0	0	13	19	38	30
ROC	6	38	0	56	0	0	50	0	50	0	25	38	12	25	0	0	88	0	12	0	19	19	0	62	0	6	68	13	13	0	6	94	0	0	0	0	31	0	69	0
Avg	7	31	5	43	14	4	36	6	40	14	23	15	14	40	8	9	30	5	40	16	18	6	14	55	7	9	44	7	23	17	6	36	2	46	10	11	26	3	43	17

Each cell in the table contains percentage values. The proportions in this table are determined based on the data presented in Table 2.2: proportion for fairness metric = # buggy cases of a metric fall into a region / # buggy cases of that metric, proportion for dataset = # buggy cases fall of a dataset into a region / # buggy cases of that dataset.

Table 2.5: Accuracy and fairness achieved by Fair-AutoML and Pafait-ML on Pafait-ML's benchmark

Data	Decision Tree				Logistic Regression				Random Forest			
	T3		PML		T3		PML		T3		PML	
	Acc	EOD	Acc	EOD	Acc	EOD	Acc	EOD	Acc	EOD	Acc	EOD
Adult	0.847	0.036	0.817	0.002	0.818	0.038	0.803	0.023	0.851	0.032	0.843	0.039
Compas	0.969	0.000	0.970	0.000	0.970	0.000	0.968	0.000	0.970	0.000	0.970	0.000

2.7.3.2 Is Fair-AutoML Effective in Fixing Fairness Bugs on Other Bias Mitigation Methods Benchmark?

Based on evaluation of Parfait-ML [59], we only use accuracy and EOD as evaluation metrics for this evaluation. To make a fair comparison with Parfait-ML, we utilize the version of *Fair-AutoML* that incorporates EOD and accuracy as its cost function (T3). The results are displayed in Table 2.5, showcasing the accuracy and bias (EOD) achieved by both *Fair-AutoML* (T3) and Parfait-ML in Parfait-ML's benchmark. The table showcases the actual results of the repaired models, rather than the difference in accuracy/fairness between the original and repaired models. Upon inspection, the results for the COMPAS dataset for both *Fair-AutoML* and Parfait-ML are similar. However, for the Adult dataset, some differences arise. For instance, with the Random Forest classifier, *Fair-AutoML* performs better than Parfait-ML in both accuracy and EOD. With the Logistic Regression classifier, *Fair-AutoML* achieved a higher accuracy but higher bias compared to Parfait-ML. Nevertheless, *Fair-AutoML* falls into the win-win trade-off region, while Parfait-ML only falls into good trade-off region (Figure 2.3). With the Decision Tree classifier, both *Fair-AutoML* and Parfait-ML fall into the win-win trade-off region (Figure 2.3); however, Parfait-ML performed better since it has lower bias. These results highlights the generalization capability of *Fair-AutoML* to repair various datasets and ML models.

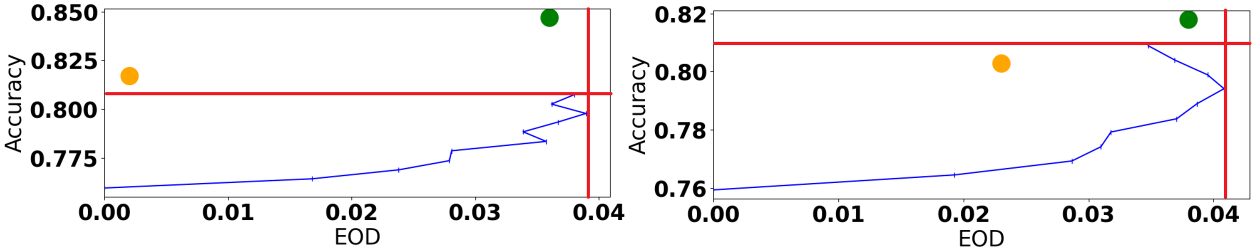


Figure 2.3: Accuracy and fairness achieved by Fair-AutoML (green circle) and Pafait-ML (orange circle) with Decision Tree (left) and logistic regression (right) on Adult dataset (Pafait-ML's benchmark). The blue line shows the Fairea baseline and red lines define the trade-off regions.

2.7.4 Ablation Study (RQ3)

We create an ablation study to observe the efficiency of the dynamic optimization function and the search space pruning separately. The ablation study compares the performance of the following tools:

- *Auto-Sklearn* (*AS*) represents AutoML.
- *Fair-AutoML* version 1 (*FAv1*) represents AutoML + dynamic optimization function.
- *Fair-AutoML* version 2 (*FAv2*) represents AutoML + dynamic optimization function + search space pruning.

To evaluate the efficiency of the dynamic optimization function, we compare the performance of *FAv1* with *Auto-Sklearn*. We compare *FAv1* with *FAv2* to observe the efficiency of the search space pruning approach. The complete result is shown in Table 2.6. Notice that we use *Fair-AutoML* to optimize different fairness metrics; thus, we only consider the metric that each tool tries to optimize. For instance, the results of Random Forest on Adult dataset in the Table 2.6 shows that achieved scores of 0.096 for DI, 0.014 for SPD, 0.024 for EOD, and 0.035 for AOD. This result means that T1 achieves 0.096 for DI, T2 achieves 0.014 for SPD, T3 achieves 0.024 for EOD, T4 achieves 0.035 for AOD. The evaluation only considers cases where the tools successfully repair the bug. The same rules described in RQ1 is applied in this evaluation.

Table 2.6: Trade-off assessment results of *Auto-Sklearn*, *FAv1*, and *FAv2*

	Metric	Model	AS	FAv1	FAv2	Model	AS	FAv1	FAv2
Adult Census	DI	RF	0.058	0.119	**0.096**	LRG	0.04	0.094	0.19
	SPD		0.016	0.053	**0.024**		0.005	0.041	0.075
	EOD		0.008	**0.015**	**0.014**		lose	0.043	0.044
	AOD		0.021	**0.025**	0.035		0.039	0.078	0.089
	DI	XGB	0.004	0.136	0.183	GBC	inv	bad	0.124
	SPD		0.003	0.046	0.074		inv	bad	0.055
	EOD		lose	0.015	0.037		inv	0.02	0.013
	AOD		0.017	0.049	0.053		**0.018**	inv	0.05
Bank Marketing	DI	RF	0.000	0.663	0.103	XGB2	lose	inv	0.158
	SPD		lose	0.026	bad		lose	inv	0.051
	EOD		lose	inv	lose		lose	lose	**0.003**
	AOD		0.016	**0.004**	0.032		lose	0.003	**0.027**
	DI	XGB1	lose	inv	0.098	GBC	lose	0.014	0.01
	SPD		lose	0.018	**0.062**		lose	lose	0.028
	EOD		lose	inv	**0.011**		lose	lose	inv
	AOD		lose	lose	**0.046**		lose	0.003	**0.025**
German Credit	DI	RF	lose	bad	bad	SVC	0.035	0.127	0.109
	SPD		lose	bad	0.052		0.021	0.078	0.092
	EOD		0.033	bad	0.045		0.068	0.112	0.101
	AOD		lose	lose	lose		lose	0.032	0.027
	DI	XGB	bad	bad	**0.07**	KNN	0.038	inv	**0.112**
	SPD		bad	lose	0.065		0.027	**0.012**	**0.075**
	EOD		0.036	lose	0.073		0.066	**0.050**	**0.085**
	AOD		lose	lose	0.037		lose	inv	0.034
Titanic	DI	RF	lose	1.04	1.549	GBC	**0.092**	0.447	0.501
	SPD		lose	0.525	0.571		inv	0.273	0.275
	EOD		lose	0.386	0.445		**0.058**	0.184	bad
	AOD		0.062	0.577	0.601		**0.058**	0.429	0.445
	DI	LRG	0.086	1.062	0.743	XGB	lose	1.205	1.364
	SPD		0.063	0.594	0.552		lose	0.314	0.542
	EOD		lose	0.556	0.557		lose	0.287	0.285
	AOD		0.101	0.651	**0.179**		lose	0.441	0.524

The data in Table 2.6 is created in the same ways as Table 2.2. For each method in the good trade-off region and win-win region (**bold number**), a trade-off measurement value is given; for other regions the region type is displayed. The values in blue, orange, and black indicate the top 1, top 2, top 3 bug fixing tools, respectively.

2.7.4.1 Are Dynamic Optimization Function and Search Space Pruning Effective in Fixing Fairness Bugs?

From Table 2.6, our results show that the dynamic optimization function approach in *Fair-AutoML* helps fix buggy models more efficiently. Comparing the performance in fixing fairness bugs, *FAv1* outperforms *Auto-Sklearn* 39 times, while *Auto-Sklearn* outperforms *FAv1* only 7 times. The search space pruning approach in *Fair-AutoML* also contributes to more efficient bug fixing, as *FAv2* outperforms both *FAv1* and *Auto-Sklearn* 46 and 55 times respectively, while *FAv1* and *Auto-Sklearn* only outperform *FAv2* 14 and 4 times respectively.

2.8 Discussion

In this work, we bring particular attention to the fairness-accuracy tradeoff while mitigating bias in ML models. Many works in the area only optimize fairness metrics by sacrificing accuracy, and do not consider the tradeoff rigorously. However, as shown by recent work [41], trivial mutation methods can also achieve fairness if accuracy is compromised in different magnitudes. Therefore, a rigorous evaluation method is necessary to demonstrate that the tradeoff is beneficial. Another limitation of existing tools is not generalizing over different ML classifiers (e.g., LRG, GBC, RF, XGB), multiple fairness metrics, and dataset characteristics. To that end, we leveraged the recent progress of AutoML in the context and achieved better tradeoff than SOTA methods. We believe that our approach is versatile and can be applied to various ML problems. Particularly, the dynamic optimization function approach remains versatile across various datasets and models. Furthermore, the search space pruning approach is refined through pre-constructed database and a matching mechanism, that capitalizes on diverse datasets stored in repositories such as *OpenML* or *Kaggle*.

We implemented *Fair-AutoML* on top of *Auto-Sklearn* to ensure its wide applicability on ML algorithms. State-of-the-art bias mitigation techniques also primarily use classic ML algorithms [6, 7, 15, 16, 19, 59] that are supported by *Auto-Sklearn*. These models are more suitable than the DL models since the fairness critical tasks in prior works commonly use tabular datasets. Should one desire to explore alternative model types not directly supported by *Auto-Sklearn*, they can adopt the general ML model adoption of *Auto-Sklearn* [27].

Our approach also outlines several opportunities towards leveraging AutoML and search-based software engineering to ensure fairness in new ML models that are becoming available. First, the greedy weight identifier algorithm's performance might suffer for complex models due to computational costs (Algorithm 1). Second, search space pruning quantitatively estimates the similarity of datasets based on data characteristics. Thus, if we do not have a dataset similar enough to the input dataset, AutoML may not perform well. To address this, we plan to regularly update our database with new datasets. Lastly, constructing suitable search spaces, particularly for resource-intensive methods like deep learning, could entail significant computational expenses.

Further works are needed to maximize the versatility and effectiveness of our approach over novel fairness-critical tasks. One key direction is to combine *Fair-AutoML* with other bias mitigation techniques, such as integrating Fair-AutoML's model with pre-processing bias mitigation methods to enhance overall pipeline fairness. Additionally, integrating *Fair-AutoML* with ensemble learning could improve both performance and fairness by capturing a broader range of biases and patterns. These directions could significantly amplify the impact of this work, making Fair-AutoML a potent tool for promoting fairness and equity in machine learning across various domains.

2.9 Threats to Validity

Construct Validity The choice of evaluation metrics and existing mitigation techniques may pose a threat to our results. We mitigate this threat by employing a diverse range of metrics and mitigation methods. First, we have used accuracy and four most recent and widely-used fairness metrics to evaluate *Fair-AutoML* and the state-of-the-art. These metrics have been commonly applied in the software engineering community [15, 16, 19, 59]. Second, we demonstrate the superiority of *Fair-AutoML* over state-of-the-art methods in different categories: pre-processing, in-processing, and post-processing, which are most advanced techniques from the SE and ML communities. For evaluating fairness and applying these mitigation algorithms except Parfait-ML [59], we have used AIF 360 toolkit. For evaluating Parfait-ML, we have used its original implementation. We create a baseline using the original Fairea implementation, enabling us to conduct a comprehensive comparison between our approach and existed mitigation methods. In the future, we intend to explore supplementary performance metrics and extend our analysis to incorporate additional mitigation techniques for a more comprehensive evaluation.

External Validity To ensure an equitable comparison with cutting-edge bias mitigation techniques, we leverage a diverse array of real-world models, datasets, and evaluation scenarios. Particularly, we utilize a practical benchmark comprising 16 real-world models thoughtfully curated by prior research [6]. Then, these meticulously chosen models undergo evaluation using four extensively studied datasets in the fairness literature [10, 60, 61]. We conducted experiments under identical setups and subsequently validated our findings [6]. In addition to assessing *Fair-AutoML* against alternative methods within our established settings and benchmarks, we subject *Fair-AutoML* to evaluation using the Parfait-ML [59] benchmark, a leading-edge bias mitigation framework.

Internal Validity Implementing *Fair-AutoML* on top of *Auto-Sklearn* may introduce a threat to its actual bias mitigation performance. In other words, the favorable outcomes achieved by *Fair-AutoML* could be attributed to its integration with *Auto-Sklearn*. To address this threat, we evaluated *Auto-Sklearn* on various benchmarks, comparing its performance with (*Fair-AutoML*) and without (*Auto-Sklearn*) our proposed approaches, to gauge the effectiveness of *Fair-AutoML*.

2.10 Conclusion

We present *Fair-AutoML*, an innovative system that enhances existing AutoML frameworks to resolve fairness bugs. The core concept of *Fair-AutoML* is to optimize the hyperparameters of faulty models to resolve fairness issues. This system offers two novel technical contributions: a dynamic optimization function and a search space pruning approach. The dynamic optimization function dynamically generates an optimization function based on the input, enabling AutoML to simultaneously optimize both fairness and accuracy. The search space pruning approach r‘educes the size of the search space based on the input, resulting in faster and more efficient bug repair. Our experiments show that *Fair-AutoML* outperforms *Auto-Sklearn* and conventional bias mitigation techniques, with a higher rate of bug repair and a better fairness-accuracy trade-off. In the future, we plan to expand the capabilities of *Fair-AutoML* to include deep learning problems beyond the scope of the current study.

Bibliography

[1] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 625–635.

[2] Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. 2023. Design by Contract for Deep Learning APIs. In *ESEC/FSE'2023: The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, California).

[3] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine bias risk assessments in criminal sentencing. *ProPublica, May* 23 (2016).

[4] Rachel KE Bellamy, Kuntal Dey, Michael Hind, Samuel C Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, et al. 2018. AI Fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *arXiv preprint arXiv:1810.01943* (2018).

[5] Reuben Binns. 2018. Fairness in machine learning: Lessons from political philosophy. In *Conference on Fairness, Accountability and Transparency*. PMLR, 149–159.

[6] Sumon Biswas and Hridesh Rajan. 2020. Do the Machine Learning Models on a Crowd Sourced Platform Exhibit Bias? An Empirical Study on Model Fairness. In *ESEC/FSE'2020: The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States).

[7] Sumon Biswas and Hridesh Rajan. 2021. Fair preprocessing: towards understanding compositional fairness of data transformers in machine learning pipeline. In *Proceedings of the*

29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 981–993.

[8] Sumon Biswas and Hridesh Rajan. 2023. Fairify: Fairness Verification of Neural Networks. In *ICSE'2023: The 45th International Conference on Software Engineering* (Melbourne, Australia).

[9] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large. In *ICSE'2022: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[10] Miranda Bogen and Aaron Rieke. 2018. Help wanted: An examination of hiring algorithms, equity, and bias. (2018).

[11] Ajay Byanjankar, Markku Heikkilä, and Jozsef Mezei. 2015. Predicting credit risk in peer-to-peer lending: A neural network approach. In *2015 IEEE symposium series on computational intelligence*. IEEE, 719–725.

[12] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data mining and knowledge discovery* 21, 2 (2010), 277–292.

[13] José P Cambronero, Jürgen Cito, and Martin C Rinard. 2020. Ams: Generating automl search spaces from weak specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 763–774.

[14] L Elisa Celis, Lingxiao Huang, Vijay Keswani, and Nisheeth K Vishnoi. 2019. Classification with fairness constraints: A meta-algorithm with provable guarantees. In *Proceedings of the conference on fairness, accountability, and transparency*. 319–328.

[15] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in machine learning software: Why? how? what to do?. In *Proceedings of the 29th ACM Joint Meeting on*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 429–440.

[16] Joymallya Chakraborty, Suvodeep Majumder, Zhe Yu, and Tim Menzies. 2020. Fairway: a way to build fair ML software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 654–665.

[17] Joymallya Chakraborty, Tianpei Xia, Fahmid M Fahid, and Tim Menzies. 2019. Software engineering for fairness: A case study with hyperparameter optimization. *arXiv preprint arXiv:1905.05786* (2019).

[18] Zhenpeng Chen, Jie M Zhang, Federica Sarro, and Mark Harman. 2022. A comprehensive empirical study of bias mitigation methods for software fairness. *arXiv preprint arXiv:2207.03277* (2022).

[19] Zhenpeng Chen, Jie M Zhang, Federica Sarro, and Mark Harman. 2022. MAAT: a novel ensemble approach to addressing fairness and performance bugs for machine learning software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1122–1134.

[20] Alexandra Chouldechova. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data* 5, 2 (2017), 153–163.

[21] Alexandra Chouldechova, Diana Benavides-Prado, Oleksandr Fialko, and Rhema Vaithianathan. 2018. A case study of algorithm-assisted decision making in child maltreatment hotline screening decisions. In *Conference on Fairness, Accountability and Transparency*. PMLR, 134–148.

[22] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O'Donoghue, Daniel Visentin, et al. 2018. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine* 24, 9 (2018), 1342–1350.

[23] Matthias Ehrgott. 2005. *Multicriteria optimization*. Vol. 491. Springer Science & Business Media.

[24] Virginia Eubanks. 2018. *Automating inequality: How high-tech tools profile, police, and punish the poor*. St. Martin's Press.

[25] Michael Feffer, Martin Hirzel, Samuel C Hoffman, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2022. An Empirical Study of Modular Bias Mitigators and Ensembles. *arXiv preprint arXiv:2202.00751* (2022).

[26] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 259–268.

[27] Matthias Feurer. 2022. Auto-Sklearn Documentation. https://automl.github.io/auto-sklearn/master/

[28] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074* (2020).

[29] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[30] Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. 2019. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the conference on fairness, accountability, and transparency*. 329–338.

[31] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. 498–510.

[32] Xuanqi Gao, Juan Zhai, Shiqing Ma, Chao Shen, Yufei Chen, and Qian Wang. 2022. FairNeuron: improving deep neural network fairness with adversary games on selective neurons. In *Proceedings of the 44th International Conference on Software Engineering*. 921–933.

[33] Giang Nguyen, Sumon Biwas, and Hridesh Rajan. 2023. *Replication Package of the ESEC/FSE 2023 Paper Entitled "Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML.* https://doi.org/10.5281/zenodo.8280911

[34] Usman Gohar, Sumon Biswas, and Hridesh Rajan. 2023. Towards Understanding Fairness and its Composition in Ensemble Machine Learning. In *ICSE'2023: The 45th International Conference on Software Engineering* (Melbourne, Australia).

[35] Przemyslaw A Grabowicz, Nicholas Perello, and Aarshee Mishra. 2022. Marrying fairness and explainability in supervised learning. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. 1905–1916.

[36] Jussi Hakanen and Joshua D Knowles. 2017. On using decision maker preferences with ParEGO. In *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 282–297.

[37] Moritz Hardt, Eric Price, and Nati Srebro. 2016. Equality of opportunity in supervised learning. *Advances in neural information processing systems* 29 (2016).

[38] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the eighth international workshop on data mining for online advertising*. 1–9.

[39] Mitchell Hoffman, Lisa B Kahn, and Danielle Li. 2018. Discretion in hiring. *The Quarterly Journal of Economics* 133, 2 (2018), 765–800.

[40] Max Hort and Federica Sarro. 2021. Did you do your homework? Raising awareness on software fairness and discrimination. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1322–1326.

[41] Max Hort, Jie M Zhang, Federica Sarro, and Mark Harman. 2021. Fairea: A model behaviour mutation approach to benchmarking bias mitigation methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 994–1006.

[42] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1946–1956.

[43] Kaggle. 2017. Adult Census Dataset. https://www.kaggle.com/datasets/uciml/adult-census-income

[44] Kaggle. 2017. Bank Marketing Dataset. https://www.kaggle.com/c/bank-marketing-uci

[45] Kaggle. 2017. German Credit Dataset. https://www.kaggle.com/datasets/uciml/german-credit

[46] Kaggle. 2017. Titanic ML Dataset. https://www.kaggle.com/c/titanic/data

[47] Faisal Kamiran and Toon Calders. 2012. Data preprocessing techniques for classification without discrimination. *Knowledge and information systems* 33, 1 (2012), 1–33.

[48] Faisal Kamiran, Asim Karim, and Xiangliang Zhang. 2012. Decision theory for discrimination-aware classification. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 924–929.

[49] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. 2012. Fairness-aware classifier with prejudice remover regularizer. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 35–50.

[50] Konstantina Kourou, Themis P Exarchos, Konstantinos P Exarchos, Michalis V Karamouzis, and Dimitrios I Fotiadis. 2015. Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal* 13 (2015), 8–17.

[51] Yanhui Li, Linghan Meng, Lin Chen, Li Yu, Di Wu, Yuming Zhou, and Baowen Xu. 2022. Training data debugging for the fairness of machine learning software. In *Proceedings of the 44th International Conference on Software Engineering*. 2215–2227.

[52] Milad Malekipirbazari and Vural Aksakalli. 2015. Risk assessment in social lending via random forests. *Expert Systems with Applications* 42, 10 (2015), 4621–4631.

[53] Giang Nguyen, Johir Islam, Rangeet Pan, and Hridesh Rajan. 2022. Manas: Mining Software Repositories to Assist AutoML. In *ICSE'22: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[54] Luca Oneto, Michele Donini, Andreas Maurer, and Massimiliano Pontil. 2019. Learning fair and transferable representations. *arXiv preprint arXiv:1906.10673* (2019).

[55] Claudia Perlich, Brian Dalessandro, Troy Raeder, Ori Stitelman, and Foster Provost. 2014. Machine learning for targeted display advertising: Transfer learning in action. *Machine learning* 95, 1 (2014), 103–127.

[56] Ralph E Steuer and Eng-Ung Choo. 1983. An interactive weighted Tchebycheff procedure for multiple objective programming. *Mathematical programming* 26, 3 (1983), 326–344.

[57] Guanhong Tao, Weisong Sun, Tingxu Han, Chunrong Fang, and Xiangyu Zhang. 2022. RULER: discriminative and iterative adversarial training for deep neural network fairness. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1173–1184.

[58] Saeid Tizpaz-Niari, Ashish Kumar, Gang Tan, and Ashutosh Trivedi. 2022. https://github.com/Tizpaz/Parfait-ML

[59] Saeid Tizpaz-Niari, Ashish Kumar, Gang Tan, and Ashutosh Trivedi. 2022. Fairness-aware Configuration of Machine Learning Libraries. *arXiv preprint arXiv:2202.06196* (2022).

[60] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 401–416.

[61] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 98–108.

[62] Rhema Vaithianathan, Tim Maloney, Emily Putnam-Hornstein, and Nan Jiang. 2013. Children in the public benefit system at risk of maltreatment: Identification via predictive modeling. *American journal of preventive medicine* 45, 3 (2013), 354–359.

[63] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rogriguez, and Krishna P Gummadi. 2017. Fairness constraints: Mechanisms for fair classification. In *Artificial Intelligence and Statistics*. PMLR, 962–970.

[64] Brian Hu Zhang, Blake Lemoine, and Margaret Mitchell. 2018. Mitigating unwanted biases with adversarial learning. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*. 335–340.

[65] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 949–960.

CHAPTER 3. DESIGN BY FAIRNESS CONTRACT FOR MACHINE LEARNING PIPELINE

Giang Nguyen[1], Shibbir Ahmed[2], Sumon Biwas[3], and Hridesh Rajan[4]

[1] Iowa State University, USA

[2] Texas State University, USA

[3] Carnegie Mellon University, USA

[4] Tulane University, USA

Modified from a manuscript to be submitted in *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*

3.1 Abstract

In the era of algorithmic decision-making, ensuring fairness while maintaining accuracy in machine learning (ML) systems is a critical challenge. To address this issue, new tools and methods are needed to detect and mitigate bias in ML-based software. While these approaches aim to fix fairness problems, they often fall short in preventing these issues in design time and informing users about them. Thus, we propose a preventive measure called a "*Fairness Contract*," which incorporates design-by-contract (DbC) principles for algorithmic fairness into the ML pipeline. Although *Fairness Contract* builds upon traditional DbC methods, we must address unique challenges. First, two types of fairness properties, hyperproperties and probabilistic properties, cannot be specified using conventional DbC mechanisms. Second, traditional DbC techniques do not support runtime assertion checking to report fairness violations in the ML pipeline. We address these issues by introducing a contract mechanism to capture fairness properties at different ML pipeline stages and report bias issues during runtime. These contracts abstract away the details of the ML algorithms, enabling systematic analysis and verification. In this work, we designed 24

contracts specifically targeting fairness bugs at various stages of the ML pipeline. These contracts were also used to evaluate our method on four fairness tasks, 45 buggy codes, and 24 correct codes. Our approach, *Fairness Contract*, is able to localize fairness bugs during runtime, which existing bug repair techniques cannot achieve. Moreover, *Fairness Contract* successfully identified 40 out of 45 buggy cases, outperforming existing fairness bug-fixing methods, which only managed to fix up to 35 out of 45 bugs. *Fairness Contract* also efficiently detects fairness bugs across various ML algorithms with a significantly shorter runtime. It only takes 1.5 seconds to detect and locate a bug, compared to the fastest existing method, which requires up to 30.9 seconds.

3.2 Introduction

Software systems have become widely utilized to make decisions, including job recommendations, employee hiring, social services, and education [14, 13, 40, 53, 38, 58, 23, 65, 26, 50, 27, 56, 4]. The widespread use of such decision-making software raises concerns about fairness issues, specifically the potential for biased decisions (fairness bugs) [8, 10]. These bugs can lead to biased decisions affecting specific groups based on protected characteristics like race, age, or gender [30, 31].

Concerns about fairness in software have been investigated by both Software Engineering (SE) [29] and Machine Learning (ML) [57] research communities. The SE community has focused on creating testing and verification methods to detect unfairness in software systems [12, 30, 2, 31, 63]. Meanwhile, researchers in the ML community have extensively studied how to develop fairness metrics and mitigation algorithms [15, 22, 28, 37, 47, 49, 66, 67]. A variety of bias mitigation techniques have been developed, including altering training data to remove bias (pre-processing) [52, 18, 17], modifying the model building process itself (in-processing) [60, 21, 41, 33, 61], or adjusting the final results to ensure fairness (post-processing) [2, 63, 69]. SE and ML literature also has witnessed a large number of recent results on testing and detecting fairness [20, 32, 64, 7, 3, 68, 68, 19]. Despite efforts to reduce and detect bias of current fairness reparing and testing methods, these methods still fall short in localizing the bias problem.

Design by Contract (DbC) is essential in the software engineering field. By defining preconditions and postconditions, DbC facilitates more thorough and targeted testing. It helps to identify incorrect assumptions and behaviors early in the development process. Therefore, we explore this approach to enhance the fairness of ML software by designing contracts that specify fairness in the ML pipeline. However, traditional DbC lacks mechanisms for documenting the fairness properties of the ML pipeline, which are essential for applying DbC to ML APIs. This paper proposes the design by contract (DbC) methodology to detect and localize fairness bugs in the ML pipeline, called *Fairness Contract*. This methodology specifies the fairness properties of the ML pipeline. This facilitates the writing of contracts—a formal agreement between a software component (such as a function, method, or class) and its clients (other components or users that interact with it)

In this work, we propose *Fairness Contract*, which comprises two major components: a fairness specification and a runtime assertion technique. Initially, *Fairness Contract* allows developers to specify hyperproperties [5, 24] and probabilistic properties of the ML pipeline, representing fairness properties. The runtime assertion technique then monitors the decision-making processes and reports any violations of these fairness expectations. Unlike existing fairness repairing methods that modify specific components of the ML pipeline to enhance fairness, our method aims to detect and locate bias issues throughout the entire ML pipeline. Different from existing fairness testing techniques that generate discrimination examples without understanding the underlying cause, our approach provides developers with insights into the root causes of fairness bugs. By specifying fairness properties that lead to unfair outcomes, we provide developers with a deeper understanding of the fairness issues.

To evaluate *Fairness Contract*, we developed 24 contracts, integrated them into ML Python software, and assessed the results using a benchmark [11]. These 24 fairness contracts were created by analyzing the strengths and weaknesses of ML algorithms from the literature [34, 11] and ML library documentation [51]. The evaluated benchmark includes four widely-used datasets from the fairness literature [31, 63, 2], 45 buggy codes, and 24 correct codes. The results demonstrate the effectiveness, efficiency, and adaptability of our approach, as *Fairness Contract* successfully detected 40 out of 45 buggy cases, surpassing the performance of existing fairness bug-fixing methods, which were only able to fix up to 35 out of 45 bugs. Additionally, *Fairness Contract* can widely detect fairness bugs in different ML algorithms with short runtime. Specifically, *Fairness Contract* takes only 1.5 seconds to detect and locate a bug, whereas the fastest existing fairness bug-fixing method requires up to 30.9 seconds.

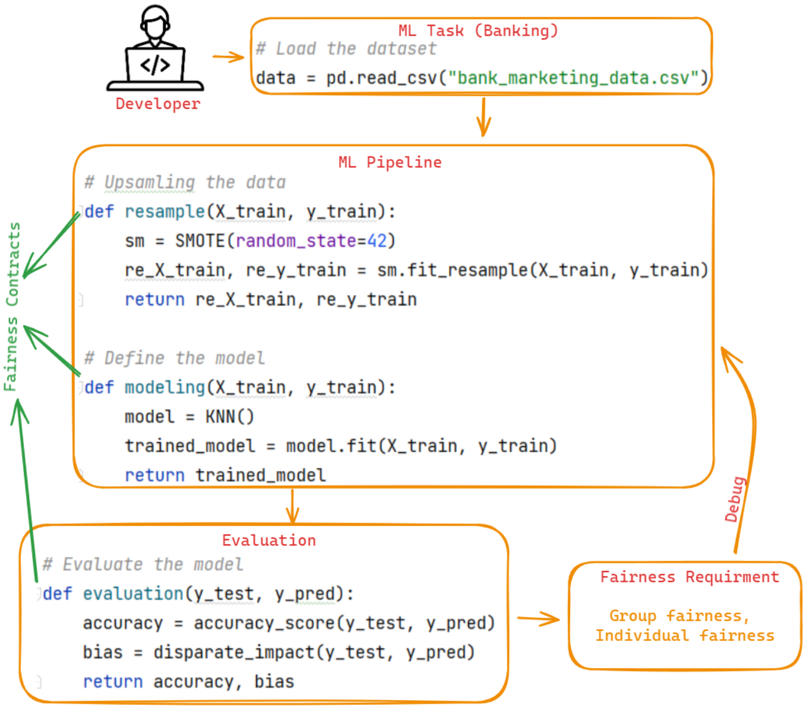
Our main contributions are the following:

- We are the first to propose writing and checking fairness contracts for ML pipelines using preconditions and postconditions to specify fairness properties.
- We introduced a fairness specification mechanism by abstracting algorithm characteristics and data properties.
- We designed and contributed 24 fairness contracts that detect and localize fairness bugs in ML pipelines.
- We designed a *Fairness Contract* [1] framework that is extensible to diverse ML algorithm types.

The paper is organized as follows: §3.3 presents a motivation, §3.4 describes the problem formulation, §3.5 indicates the guidelines for using *Fairness Contract*, §3.6 shows the *Fairness Contract* approaches, §3.7 presents the our evaluation, §3.8 discusses the limitations and threats to validity of *Fairness Contract*, §3.9 concludes.

3.3 Motivation

Figure 3.1 illustrates a scenario where software developers aim to create an ML system that adheres to ensure the *individual fairness* assumption while predicting the success of bank telemarketing. Throughout the procedure's implementation and deployment phases, the developers need to verify that their fairness assumptions are valid. In practice, ensuring these assumptions



```
# Load the dataset
data = pd.read_csv("bank_marketing_data.csv")
```

```
# Upsamling the data
def resample(X_train, y_train):
    sm = SMOTE(random_state=42)
    re_X_train, re_y_train = sm.fit_resample(X_train, y_train)
    return re_X_train, re_y_train

# Define the model
def modeling(X_train, y_train):
    model = KNN()
    trained_model = model.fit(X_train, y_train)
    return trained_model
```

```
# Evaluate the model
def evaluation(y_test, y_pred):
    accuracy = accuracy_score(y_test, y_pred)
    bias = disparate_impact(y_test, y_pred)
    return accuracy, bias
```

Figure 3.1: Motivating Example for Fairness Contracts

hold presents two main challenges. First, the developers cannot confirm whether the system satisfies the fairness assumptions until it completes execution. Second, if the system violates these fairness assumptions, the developers must review the system to identify the issue. Given the complexity of ML systems, which consist of multiple components, it can be challenging to pinpoint the source of the problem. Consequently, the developers must inspect each component individually and rerun the entire system to determine if the issue has been resolved, making this a time-consuming task. Unfortunately, no method is currently available to help developers address this problem. The current fairness repairing and testing mechanisms take a "build-then-fix" strategy, which does not provide design time capability to ensure fairness.

To enable the developer to specify fairness more efficiently, we propose "*Fairness Contract*," which involves incorporating design-by-contract principles into the ML pipeline to ensure fairness. Similar to traditional design-by-contract principles [39], the client must guarantee certain conditions before calling a method defined by the class. In return, the class guarantees specific properties will hold after the call. What is novel about *Fairness Contract* is its ability to describe the fairness behavior exhibited by individual components within the ML pipeline in isolation. This capability allows for a detailed understanding of the fairness and accuracy implications when implementing modifications within the ML pipeline. Thus, *Fairness Contract* is able to detect fairness violations immediately as they occur during the execution of the ML program. Consequently, the modularity of *Fairness Contract* allows us to pinpoint the exact location of fairness violations in ML software. Moreover, since the program terminates immediately upon detecting a violation, this approach saves considerable time and computational resources by eliminating the need to run the entire system to identify issues. *Fairness Contract* allows developers to add contracts to each component of an ML pipeline. So, when a contract is violated, the developer can immediately identify where the problem occurs.

3.4 Problem Formulation

3.4.1 Classification Program

Our goal is to propose a programming language approach to fairness specification, where the developers declaratively state fairness requirements for their program $M_\lambda : X \rightarrow y$ that maps a given member $x \in X$ (e.g., a person's demographic information) to a single binary output $y = \{0, 1\}$ (e.g., whether to approve a loan). For instance, M_λ might represent an ML pipeline with a set of hyperparameters λ. Our method requires only black-box access to M_λ, allowing the developer to specify fairness based on any chosen input $x \in X$ and the corresponding output $\hat{y} = M_\lambda(x)$.

3.4.2 ML Pipeline

At the heart of effective ML software lies the ML pipeline, a structured sequence of processes designed to streamline the development and deployment of machine learning models. An ML pipeline typically encompasses several key stages: data preprocessing, feature engineering, and modeling. Data preprocessing involves cleaning and transforming raw data into a usable format. Tasks include handling missing values, normalizing data, and encoding categorical variables. Feature engineering is used to select, create, or modify to improve the model's performance.

Modeling involves selecting and training machine learning models using preprocessed data and engineered features. This includes choosing appropriate algorithms, tuning hyperparameters, and evaluating model performance to ensure it meets the desired criteria. We have designed 24 fairness contracts based on these three components of the ML pipeline.

3.4.3 Fairness Specification

In this section, we categorize and describe different fairness specifications that can be formalized in our fairness contract. We first establish some notation. We are addressing a problem in which each individual in the population has a true label in $y = \{0, 1\}$. We assume the presence of a protected attribute $z = \{0, 1\}$, such as race, sex, or age, where one label is considered privileged (denoted as 0) and the other is unprivileged (denoted as 1). The predictions are $\hat{y} \in \{0, 1\}$ that should not only be correct in relation to the actual outcome y but also be unbiased towards a specific protected characteristic z. We have collected the most concerning fairness requirements from the recent works and created this set of fairness specifications that the user can define using *Fairness Contract*. Users can add additional fairness specifications as needed.

3.4.3.1 Group Fairness

Group fairness ensures different demographic groups receive equitable treatment and outcomes. It focuses on preventing bias and discrimination against specific groups based on characteristics such as race, gender, age, or socioeconomic status. We describe how four fairness specifications from the machine learning literature can be formalized in our specification language.

The *Disparate Impact (DI)* measures fairness by comparing the proportion of the unprivileged group that receives a favorable label to the proportion of the privileged group that receives the same label [28, 66]. The DI value ranges from 0 to 1, with higher values indicating better fairness. Given $\beta \in [0, 1]$, we can specify the DI property as follows:

$$DI = \frac{Pr[\hat{y}=1|z=0]}{Pr[\hat{y}=1|z=1]} \geq \beta$$

The *Statistical Parity Difference (SPD)* quantifies the disparity between the probability of receiving a favorable label for the unprivileged group and that for the privileged group [15]. The SPD ranges from -1 to 1, with values closer to 0 indicating better fairness. Given $\beta \in [0,1]$, we can specify the SPD property as follows:

$$SPD = |Pr[\hat{y} = 1|z = 0] - Pr[\hat{y} = 1|z = 1]| \leq \beta$$

The *Equal Opportunity Difference (EOD)* measures the disparity between the true-positive rates of the unprivileged and privileged groups. The EOD ranges from -1 to 1, with values closer to 0 indicating better fairness. Given $\beta \in [0,1]$, $TPR_u = Pr[\hat{y} = 1|y = 1, z = 0]$ and $TPR_p = Pr[\hat{y} = 1|y = 1, z = 1]$, we can specify the EOD property as follows:

$$EOD = |TPR_u - TPR_p| \leq \beta$$

The *Average Absolute Odds Difference (AOD)* is the mean difference between the true-positive rate and the false-positive rate of the unprivileged and privileged groups [37]. Similar to EOD, AOD ranges from -1 to 1, with values closer to 0 indicating better fairness. Given $\beta \in [0,1]$, $FPR_u = P[\hat{y} = 1|y = 0, z = 0]$, and $FPR_p = P[\hat{y} = 1|y = 0, z = 1]$, we can specify the average AOD property as follows:

$$AOD = \frac{1}{2} * (|FPRu - FPRp| + |TPR_u - TPR_p|) \leq \beta$$

To use all the metrics in the same setting, DI has been plotted in the absolute value of the log scale, and SPD, EOD, AOD have been plotted in absolute value [18, 42]. Thus, the bias score of a model is measured from 0, with lower scores indicating more fairness.

3.4.3.2 Individual Fairness (IF)

Suppose D is a dataset containing n data instances, where each instance $x = (x_1, \ldots, x_n)$ is a tuple with n attribute values. The set of attributes is denoted by $A = A_1, \ldots, A_n$ with its domain I, where $A_i = a|a \in I_i$. The set of protected attributes is denoted by P where $P \subset A$. Given $\alpha \geq 0$, we can verify individual fairness by ensuring that there is no pair of data instances (x, x') in the input domain such that:

$$IF = \begin{cases} |x_i - x'_i| \leq \beta, \forall i \in A \\ x_j \neq x'_j, \exists j \in P \\ M_\lambda(x) \neq M_\lambda(x') \end{cases}, Pr[IF] \leq \beta$$

3.4.3.3 Data properties

Real-world ML software is complex, as it is built using a variety of ML algorithms. Each algorithm has its own strengths and weaknesses, which impact the overall fairness of the program. Despite the complexity of ML software, there are common fairness properties that can be extracted from the characteristics of these algorithms, such as outliers, imbalance ratio [11], as following:

Imbalance ratio can significantly impact fairness in machine learning models, leading to biased outcomes that disadvantage certain groups. For instance, a high imbalance ratio can cause the model to learn skewed decision boundaries, making it harder to classify instances of the minority class correctly. We can specify the imbalance ratio property as follows:

$$IR = \frac{N_{major}}{N_{minor}} \leq \beta$$

Where N_{major} and N_{minor} are the number of instances in the majority class and the minority class, respectively.

Feature collinearity occurs when two or more features are highly correlated. This means that one feature can be linearly predicted from the others. The problem becomes particularly concerning when at least one feature is highly related to a protected attribute (e.g., race, gender, age). High collinearity with a protected attribute can pose several fairness issues, including biased model predictions and discrimination. We can specify the feature collinearity with a protected attribute using pearson correlation coefficient (PCC) [25] as follows:

$$PCC = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2 \sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \leq \beta$$

Where, X_i and Y_i are the individual sample points, $\bar{X}$ and $\bar{Y}$ are the means of X and Y, n is the number of data points.

3.5 Guidelines for Writing, Collecting, and Using Fairness Contracts

3.5.1 Writing Fairness Contracts

Similar to the design by contract method [54], *Fairness Contract* employs an annotation-based approach to integrate contracts into various stages of the ML pipeline. The client is required to ensure specific conditions before invoking a method defined by the class (*preconditions*), and in return, the class guarantees certain properties that will hold after the call (*postconditions*). What is novel about *Fairness Contract* is its ability to address the challenge of specifying fairness within the ML pipeline and to facilitate developers in writing contracts for the ML pipeline. In the following sections, we will demonstrate the syntax of *Fairness Contract* and illustrate how to write *Fairness Contract* in an ML pipeline.

3.5.1.1 Syntax

Fairness Contract comprises two main components used to annotate APIs: @fairness_contract and @new_contract. The annotation @fairness_contract allows developers to register new contracts, while @new_contract enables them to write functions for performing computations necessary for a contract, as well as for verifying preconditions and postconditions. The primary function of *Fairness Contract* is to extract and verify fairness properties such as group fairness, individual fairness, and data properties. Additionally, *Fairness Contract* is not limited to checking simple data types like strings, floats, numbers, arrays, and booleans. It also incorporates logical operators to support arithmetic and comparison expressions like AND (,) and OR (|).

3.5.1.2 Illustrative Example

Example 3.1 shows an example, where the developers use @fairness_contract to for checking postconditions using returned values of ”prediction” functions (lines 21-22). Subsequently, the developers employ @new_contract to capture the necessary fairness properties for the defined contract (lines 1 and 11). In the @new_contract annotation, developers can use predefined functions (line 13) or write their own functions to obtain fairness properties (line 4). Example 3.2

presents another example where developers use @fairness_contract to check preconditions using the input values of ”modeling” functions (line 12).

3.5.2 Collecting Fairness Contracts

To show the effectiveness of *Fairness Contract* in specifying fairness for the ML pipeline, we created a list of 24 contracts. This process began with a comprehensive review of existing machine learning libraries, such as Scikit-learn [51], along with relevant state-of-the-art research articles [34, 11]. We identified key factors that contribute to bias in machine learning algorithms, such as outliers for scaling algorithms, data imbalance for sampling algorithms, and feature collinearity for feature selection algorithms [11, 17, 51]. By analyzing the strengths and weaknesses of various ML algorithms documented in the literature, we extracted critical insights into their performance and fairness properties. Using this information, we compiled a set of fairness contracts outlining ML algorithms’ expected behavior and performance criteria, ensuring they adhere to fairness standards and effectively mitigate bias. This list serves as a guideline for evaluating and improving the fairness of machine learning models across diverse applications. Table 3.1 presents a detailed list of fairness contracts.

3.5.3 Using Fairness Contracts

Although *Fairness Contract* is easy for developers to write due to its simple syntax, selecting the correct conditions for different ML algorithms in the pipeline can be challenging since the diverse range of algorithms may each require capturing different fairness properties. To assist developers, we aim to provide guidance based on key fairness requirements from the literature, which can be specified using *Fairness Contract*.

3.5.3.1 Data preprocessing

Sampling selects a representative data subset to reduce the computational load or balance class distributions, but it can introduce bias, especially with upsampling techniques [11, 17]. Checking

Table 3.1: Fairness Contracts

		Fairness Type	Fairness Contract	
Data Preprocessing	Sampling	Group	Precondition 1: imbalance_ratio $\leq \beta$	Modular Fairness
		Group	Precondition 2: sampling = upsampling	
		Group	Postcondition 1: imbalance_ratio(y) = 0	
		Indiv	Postcondition 2: $Pr[IF] \geq \beta$	
	Scaler	Group	Precondition: kurtosis(x_train, y_train) $\leq \beta$	
		Group	Postcondition: similarity(x_train, x_train') $> \beta$	
	Imputation	Group	Precondition: imputation_type = drop	
		Indiv	Postcondition: $Pr[IF] \geq \beta$	
FE	Feature-Selection	N/A	Precondition: True	
		Group	Postcondition 1: PCC $\leq \beta$	
		Group	Postcondition 2: num_features $\leq \beta$	
Classifier	Distance-based	Group	Precondition: kurtosis(x_train, y_train) $\leq \beta$	
		Group	Precondition: imbalance_ratio $\leq \beta$	
		N/A	Postcondition: True	
	Linear	Group	Precondition: kurtosis(x_train, y_train) $\leq \beta$	
		N/A	Postcondition: True	
	Tree-based	Group	Precondition: imbalance_ratio $\leq \beta$	
		N/A	Postcondition: True	
Pipeline		N/A	Precondition: True	Global Fairness
		Group	Postcondition 1: test_bias $\leq \beta$	
		Group	Postcondition 2: test_accuracy $\geq \beta$	
		Group	Postcondition 3: test_bias = 0. test_accuracy >max(P(Y = 1), P(Y = 0))	
		Indiv	Postcondition 4: $Pr[IF] \leq \beta$	
		Indiv	Postcondition 5: D(M(x), M(x')) >$\|xi - x'i\|$	

FE represents Feature Engineering.

the label ratio is essential before applying sampling. While upsampling addresses imbalance, it can create unwanted samples, causing individual bias. Thus, developers should validate individual fairness after using upsampling algorithms.

Scaler adjusts feature magnitudes to normalize or standardize data for better ML model performance. However, algorithms like standard scaler or max-abs scaler are sensitive to outliers [51, 11]. Therefore, developers should add a precondition to check whether the data contains a significant number of outliers and extreme values.

Imputation replaces missing data to maintain dataset integrity. Removing samples with missing values can introduce bias, so developers should avoid this [11]. Moreover, imputation techniques that fill missing values with constants can introduce individual bias problems. Thus, developers should add a postcondition to validate individual fairness after using these imputation methods.

3.5.3.2 Feature engineering

Feature selection identifies relevant features to enhance model performance by reducing dimensionality and overfitting. However, it can introduce bias if it drastically reduces features or retains those highly collinear with protected attributes [11]. Developers should check the number of remaining features and their collinearity with protected attributes as postconditions.

3.5.3.3 Modeling

Distance-based classifier predicts the class of a data point based on the distances to points in the training set, such as k-nearest neighbors (KNN). These classifiers are sensitive to outliers and extreme values [51]. Therefore, developers should include a precondition to verify the presence of outlier values before using these algorithms.

Linear classifier separates classes with a linear decision boundary, including algorithms like logistic regression. This type of classifier is sensitive to a high number of multicollinear features, as well as outliers values [51, 11]. Therefore, developers should include a precondition to check for these properties before using the classifier.

Tree-based classifier uses decision trees to split data into subsets based on feature values, with popular examples including decision trees or random forests. Nevertheless, tree-based classifiers perform poorly with imbalanced data [51]. Therefore, developers should add preconditions to validate the imbalance ratio before using a tree-based classifier.

3.6 Fairness Contracts

This section details the key contributions of *Fairness Contract*. *Fairness Contract* enables ML developers to specify fairness in ML pipelines by establishing global fairness criteria and defining fairness properties for different components. This helps localize fairness issues. *Fairness Contract* also supports runtime-checking, incrementally verifying fairness criteria for each module as decisions are made.

Algorithm 4 Runtime Assertion Checking for Probabilistic Properties

```
Input: An input $X$ with $n$ samples: $\{x_1, x_2, \ldots, x_n\}$
Output: None
function ATTR($x, r$)                    ▷ Define attributes extraction function
    ...                                  ▷ Developer decide how to extract necessary attributes
    return attr                          ▷ Return list of attributes
function VAL($prop$)                     ▷ Define validation function
    ...                                  ▷ Developer defines how to validate updated property
    return boolean value
hist ← []                                ▷ Initialize history list
for each $x \in X$ do
    $r \leftarrow M_\lambda(x)$
    attr ← ATTR($x, r$)
    hist.append(attr)                    ▷ Add current attr to history
    p ← PROPERTY(attr)                   ▷ Update old fairness property
    $val \leftarrow$ VAL(p)
    if $val$ = False then
        Raise alert message
        Abort the program
```

3.6.1 Global Fairness Contracts

In practical scenarios, software developers who design decision-making processes may want to ensure that their software meets certain fairness criteria, such as group fairness or individual fairness (global fairness requirements) [18, 60, 2, 55]. To support this goal, we propose *Fairness Contract*, which incorporates fairness definitions as fundamental elements of a programming language. This would enable developers to explicitly indicate in their code that a particular procedure (an ML pipeline) complies with a specified fairness concept. As the procedure executes and makes decisions, the runtime system continuously monitors these decisions and notifies the developer if any fairness problem occurs.   The issue is that post-training fairness properties, such as group and individual fairness, are typically probabilistic properties and hyperproperties [5, 24], cannot be specified using conventional DbC techniques. Standard contracts only enforce constraints on an API method's input and return values. Thus, verifying global fairness violations cannot be done through a single execution as in standard DbC. Instead, we need to observe the decisions produced by the procedure and utilize statistical tools to conclude with reasonably high confidence that a fairness property is not being upheld.

```
@new_contract
def individual_fairness(X, y_pred):
    def IF(X, y_pred) -> bool:
        "Developer verify individual fairness here"
    if not IF:
        raise ContractException("Violation: Failure to
        ensure individual fairness")
@new_contract
def disparate_impact(y_pred):
    DI = disparate_impact_ratio(y_pred)
    threshold = 0.8
    if DI < threshold:
        raise ContractException("Violation: disparate
        impact is lower than the threshold." *
        str( '%.2f' % threshold))
@fairness_contract(return= 'disparate_impact')
@fairness_contract(return= 'individual_fairness')
def prediction(model, X):
    y_pred = model.predict(X)
    return y_pred
```

Listing 3.1: Global Fairness Contract Examples

To address this issue, we developed a method for writing global fairness contracts using functions encompassing various fairness definitions from the literature. These functions also enable the evaluation of other performance metrics for machine learning software, such as accuracy and F1 score. Example 3.1 illustrates an example of specifying global fairness requirements during the deployment phase. The developers aim to ensure that their ML pipeline maintains a selection rate for minorities at least 0.8 times that of non-minorities and verifies individual fairness. To achieve this, the developers specify a postcondition on the "prediction" method using the @fairness_contract annotation. They then use the @new_contract annotation to capture fairness properties, such as disparate impact or verifying individual fairness. With @new_contract, developers can specify conditional expectations (group fairness) or hyperproperties (individual fairness), where comparisons are made between pairs of input samples.

Algorithm 5 Runtime Assertion Checking for Hyperproperties

```
Input: an input $X$ with $n$ samples: $\{x_1, x_2, \ldots, x_n\}$
Output: None
function VAL($[x, r], [x', r']$)                              ▷ Define validation function
    ...                                 ▷ Developer define how to validate two pair of input
    return boolean value
hist ← []                                                      ▷ Initialize history list
for each $x \in X$ do
    $r \leftarrow M_\lambda(x)$
    for each $[x', r'] \in$ hist do
        $val \leftarrow$ VAL($[x, r], [x', r']$)
        if $val$ = False then
            Raise alert message
            Abort the program
    hist.append($[x, r]$)                               ▷ Add current $[x, r]$ to history
```

3.6.2 Decomposing Global Fairness Contract into Modules

By specifying only global fairness, developers may struggle to identify the source of the problem since the problem can stem from different stages in the ML pipeline (discussed in Section 3.5.3). To address this problem, *Fairness Contract* allows users to decompose global fairness requirements into modules, enabling developers to specify fairness properties for different ML pipeline components. These modular fairness properties are then integrated to ensure global fairness. Enforcing fairness contracts for different ML pipeline components requires a mechanism to capture data characteristics beyond just the formal parameters and return values of the ML APIs. However, within the diverse range of ML algorithms, such as standard scaler (data preprocessing), PCA (feature engineering), logistic regression (modeling), etc., we also need to specify a wide variety of data characteristics. For example, to specify fairness properties for scaler algorithms, we might need to capture data distribution properties (as shown in Example 3.2). Particularly, scaler algorithms are used to adjust the scale of data, transforming it to fit within a specific range or distribution. For example, algorithms like the standard scaler or min-max scaler adjust extreme values to a different value range. However, when features like "salary" are scaled, their significance to the model's prediction may decrease. Initially, extreme values, such as very high or very low salaries, significantly impact the model's decisions. These extreme values are essential features that help the model achieve high accuracy and reduce bias. Thus, data distribution is an important fairness property that developers should observe before using scaler algorithms.

```
@new_contract
def kurtosis(X_train, y_train):
    mfe = pymfe.mfe.MFE().fit(X_train, y_train)
    ft = mfe.extract(features=["kurtosis"])
    threshold = 10
    if ft > threshold:
        msg = "Violation: Distance-based classifier
              is sensitive to outliers and extreme
              values, which are prevalent in the dataset."
        raise ContractException(msg)
@fairness_contract('kurtosis')
def modeling(X_train, y_train):
    model = KNN()
    trained_model = model.fit(X_train, y_train)
    return trained_model
```

Listing 3.2: Modular Fairness Contract Examples

These data properties often require complex computations to capture. For instance, specifying individual fairness requires access to the value of every attribute and the label of the data. These complex computations cannot be specified using conventional DbC techniques. To address this problem, *Fairness Contract* allows developers to specify fairness requirements by using predefined functions or by writing their own functions to measure data properties. To illustrate the ability of our method, we collected 24 fairness contracts. This required us to extensively study related empirical research [11, 34], conduct experiments, and extract the strengths and weaknesses of ML algorithms from an ML libraries [51]. Examples 3.2 and 3.1 demonstrate how developers can apply these contracts to specify fairness. In this solution, by using the @new_contract annotation, the developer can use a predefined function, such as “kurtosis” from the “Pymfe” library [6], as a precondition to capture the data distribution of the input data. Alternatively, the developer can also write their own functions to verify individual fairness. This is how the *Fairness Contract* mechanism enables specifying and checking modular fairness contracts on ML pipeline components, helping developers save time and computational resources in debugging.

3.6.3 Fairness Contract Runtime Assertion Technique

Many fairness properties can be obtained and verified during the runtime of an ML pipeline. Therefore, we develop a runtime-checking technique that incrementally verifies provided fairness specifications each time a decision is made. To illustrate the practical application of our contribution, consider a scenario where a machine learning (ML) pipeline is deployed to decide whether individuals should receive loans. In such a situation, the deployed ML pipeline may receive a set of loan applicants as input, $x_1, x_2, ..., x_n$. Each time the ML pipeline processes an input to make a prediction, the developer can specify fairness properties (e.g., group fairness or individual fairness) to check if the pipeline violates these fairness specifications with each decision. This approach allows developers to identify potential issues in the deployed ML pipeline promptly, without needing to complete a full run of all given inputs. To illustrate how *Fairness Contract* can assist developers in achieving this goal, we present the following two examples.

The first example involves a developer specifying that the deployed ML pipeline must satisfy the definition of individual fairness, as shown in Example 3.1. In this scenario, the developer registers new contracts (line 1) and then parses and validates these newly defined contracts as postconditions applied to the functions (line 17). Algorithm 5 demonstrates how a developer can perform runtime assertions for specifying hyperproperties, such as individual fairness, which involve comparing multiple program executions of an ML pipeline “M_λ”. Specifically, we first define a list to store the history of previous executions, including previous inputs and results (line 5). For each input “x”, we compare “x” and the result “M_λ(x)” with the previous inputs and results (lines 6-9). At line 9, we call a function “VAL” to validate the new inputs and results against the old ones. The implementation of the “VAL” function is not specified, as it depends on the developer’s requirements. For instance, the “VAL” function could be used to calculate the similarity between two pairs of samples. Finally, if the fairness requirement is not satisfied, an alert message will be raised, and the program will be aborted.

In the second example, the developer specifies that the deployed ML pipeline must satisfy the definition of group fairness, as shown in Example 3.1. Like the previous example, the developer registers new contracts (line 8) and defines these contracts as postconditions applied to the functions (line 16). Algorithm 4 demonstrates how a developer can perform runtime assertions for this case. Instead of storing all previous inputs and execution results of the program like Algorithm 5, we only need to extract the necessary attributes, such as the current TPR_u, TPR_p, etc. (lines 9-13), which are used to capture the fairness properties (line 14). We then validate the updated fairness property using a function "VAL". In this case, we do not specify the implementation of "VAL", as it depends on the developer's requirements. For instance, the developer can validate the fairness requirement of the function "M_λ" using concentration inequalities [5]. Finally, we will raise an alert message and terminate the program if the fairness requirement is not satisfied.

3.7 Evaluation

This section describes the experiment setup to evaluate the effectiveness, efficiency, and applicability of *Fairness Contract* in detecting and localizing fairness performance bugs. We begin by stating research questions and then address the questions focusing on the effectiveness, efficiency, adaptability, and overhead of *Fairness Contract*.

RQ1 (Effectiveness): How effectively does Fairness Contract detect fairness bugs?

RQ2 (Efficiency): How efficient is *Fairness Contract* for detecting fairness bugs in terms of execution time?

RQ3 (Applicability): Is *Fairness Contract* enabled to find fairness performance bugs in Python programs?

3.7.1 Experiment Setup

To evaluate *Fairness Contract* in Python, we extended *PyContracts* [36] to give users the ability to specify fairness in ML pipelines and enhance overall usability. Our experiments were conducted using Python 3.8, on a system equipped with two 18-core Intel Skylake 6140 CPUs.

Table 3.2: The datasets used in the fairness bug detection experimentation.

Dataset	Size	# F	PA	# B	# C	Description
Adult Census [43]	32,561	12	race, sex	9	10	Adult Cencus contains the financial information of individuals from the 1994 U.S. census, which isused to predict whether an individual earns an annual income greater than $50K.
Bank Marketing [44]	41,188	20	age	13	3	Bank Marketing includes demographic, financial, and social information on direct marketing campaigns of a Portuguese banking institution that is used to identify whether the client will subscribe to a term deposit.
German Credit [45]	1,000	21	age, sex	16	3	German Credit is used to predict credit risk levels of individual based on their demographic and credit information.
Titanic [46]	891	10	sex	7	8	Titanic contains individual information of Titanic passengers used to predict who survived the Titanic shipwreck.

# F: Feature count, PA: Protected attribute, # B: Buggy case count, # C: Correct case count.

3.7.1.1 Benchmark

We evaluated our method using real-world programs collected from a recent work [11], with our benchmark consisting of 45 buggy cases and 24 correct cases collected from Kaggle covering different ML pipeline structures and including different types of ML pipeline component. These programs have been created for four popular datasets, which are popular in fairness literature [13, 62, 63]. Table 3.2 shows the details about these datasets. In this work, we have generated the fairness contracts for three main components of the ML pipeline: data preprocessing, feature engineering, and modeling. The ML APIs, which have been validated using our contracts, are listed in our artifact [1]

3.7.1.2 Evaluated Learning Techniques

To evaluate the efficiency and effectiveness of *Fairness Contract*, we compare it with various fairness bug-fixing methods, including: *Reweighing* (**R**) [47], *Disparate Impact Remover* (**DIR**) [28], *Parfait-ML* (**PML**) [61], *Equalized Odds* (**EO**) [37], *FaX-AI* (**FAX**) [35], *Reject Option Classification* (**ROC**) [48]. These methods are chosen because most fairness bugs are performance-related (e.g., low accuracy, high bias), and our fairness contracts focus on addressing these issues. Thus, it is fair to compare *Fairness Contract* with methods targeting performance-related bugs.

These fairness bug-fixing methods aim to enhance various fairness criteria, such as Disparate Impact (DI) and Average Odds Difference (AOD), by identifying and correcting fairness bugs. To ensure a fair evaluation between *Fairness Contract* and fairness bug-fixing techniques, we assume that a technique will consider a program to have a fairness bug if it can improve all the targeted fairness metrics. Conversely, if the technique cannot improve all the targeted metrics, it will consider the program free of fairness bugs. For instance, PML focuses on improving fairness in terms of Equal Opportunity Difference (EOD) and AOD. If PML successfully improves both metrics for a program, it considers the program to have a fairness bug. However, if PML can improve only one of these metrics, it considers the program to be free of fairness bugs. In this work, we do not compare *Fairness Contract* with fairness testing methods [20, 32, 64, 7], as to the best of our knowledge, no fairness testing methods aim at detecting performance bugs in ML pipelines. Most recent fairness testing methods focus on generating discrimination testing samples to identify individual fairness bugs, which is not the primary focus of our paper.

3.7.1.3 Evaluation Configuration

In this evaluation, the mean performance of each method is calculated as the average of 10 runs, a commonly used practice in fairness literature [9, 10, 16]. Specifically, we ran each method 10 times to collect performance data, including fairness metrics (DI, SPD, EOD, AOD) and runtime. The mean values of these metrics and runtime are then used in our evaluation. The mean fairness metrics are also used to count the number of bugs detected by the fairness bug-fixing method.

3.7.2 Effectiveness (RQ1)

To assess the effectiveness of *Fairness Contract*, we measure the number of bugs detected by *Fairness Contract* compared to other bias mitigation techniques. Specifically, we designed 24 fairness contracts applicable to different ML pipeline components and algorithms. We evaluate *Fairness Contract* by comparing its performance with 7 fairness bug-fixing techniques across four fairness datasets, 45 buggy cases, and 24 correct cases.

Table 3.3 shows *Fairness Contract* detected 40 out of 45 fairness bugs (89%), while the best bias mitigation techniques detected up to 35 out of 45 bugs (78%). This demonstrates *Fairness*

Table 3.3: Comparison of number of detected bugs between *Fairness Contract* and fairness bug fixing techniques

ML Component	API Type	R	DIR	PML	EO	FAX	CEO	ROC	FC
Data Preprocessing	**Encoder (1)**	0 (0.00)	0 (0.00)	1 (1.00)	1 (1.00)	0 (0.00)	0 (0.00)	1 (1.00)	1 (1.00)
	Sampling (15)	2 (0.13)	5 (0.33)	13 (0.87)	14 (0.93)	4 (0.27)	10 (0.67)	5 (0.33)	15 (1.00)
	Scaler (6)	2 (0.33)	1 (0.17)	5 (0.83)	6 (1.00)	0 (0.00)	1 (0.17)	2 (0.33)	6 (1.00)
	Imputation (2)	2 (1.00)	1 (0.50)	2 (1.00)	2 (1.00)	0 (0.00)	2 (1.00)	0 (0.00)	2 (1.00)
Feature Engineering	**Feature Selection (11)**	3 (0.27)	2 (0.18)	8 (0.72)	6 (0.55)	4 (0.37)	3 (0.27)	3 (0.27)	9 (0.82)
Classifier	**Distance-based (4)**	0 (0.00)	1 (0.25)	1 (0.25)	1 (0.25)	0 (0.00)	1 (0.25)	0 (0.00)	4 (1.00)
	Linear (3)	0 (0.00)	0 (0.00)	3 (1.00)	3 (1.00)	0 (0.00)	1 (0.33)	3 (1.00)	3 (1.00)
	Tree-based (3)	1 (0.33)	1 (0.33)	1 (0.33)	2 (0.67)	2 (0.67)	2 (0.67)	0 (0.00)	0 (0.00)
Total (45)		10 (0.22)	11 (0.24)	34 (0.76)	35 (0.78)	10 (0.22)	20 (0.44)	14 (0.31)	40 (0.89)

The 2^{nd} column lists ML algorithm types and their total buggy cases. Columns 3^{th} to 10^{th} display "# detected bugs (accuracy)".

Contract's superiority in detecting fairness bugs. *Fairness Contract* can detect more fairness bugs because it is comprehensive enough to identify fairness issues across different types of ML algorithms. On the other hand, current fairness bug-fixing techniques only focus on one ML pipeline component, such as data preprocessing or modeling, which limits the number of algorithms they can support. This explains why they detect fewer bugs. For example, a common fairness issue with the K-nearest-neighbor (KNN) algorithm arises when developers use it without rescaling the data, leading to biased predictions. This bias occurs because KNN relies on distance calculations, and features with larger numerical ranges can dominate, skewing results. Therefore, scaling algorithms should be used before applying KNN. *Fairness Contract* can detect this problem by checking for varying numerical ranges in the input data for KNN. However, existing fairness bug-fixing techniques are not comprehensive enough to address this issue. They often focus on finding bugs within the KNN algorithm itself, such as tuning hyperparameters, which may not resolve the underlying problem of data scaling.

According to 3.3, *Fairness Contract* has lower detection accuracy in tree-based algorithms because the bugs are due to incorrect hyperparameter selection. However, we currently do not have fairness contracts to detect bias due to hyperparameter selection. In contrast, most of the evaluated fairness repairing techniques address bias by modifying hyperparameters, explaining their better performance. However, these results do not indicate that *Fairness Contract* is ineffective at detecting bias due to hyperparameter selection. We believe that our method could perform well in addressing this issue if we have a comprehensive set of fairness properties tailored to this problem. Additionally, *Fairness Contract* achieved an 82% detection rate in feature selection cases because developers often use custom code instead of predefined functions, making it harder to detect fairness bugs with contracts.

Table 3.4: Effectiveness of *Fairness Contract*

Dataset	**FP**	**TP**	**FN**	**TN**	**Precision**	**Recall**	**Accuracy**
Adult Census	0	8	1	8	1.00	0.89	0.94
Bank Marketing	1	13	0	2	0.93	1.00	0.94
German Credit	2	10	0	7	0.83	1.00	0.89
Titanic	2	9	1	5	0.82	0.90	0.82

To evaluate the effectiveness of *Fairness Contract* in detecting fairness bugs, we measure its precision, recall, and accuracy across four datasets. Notably, as shown in 3.4, *Fairness Contract* achieves the highest precision on the Adult Census dataset (100%), the highest recall on both the Bank Marketing and German Credit datasets (100%), and the highest accuracy on the Adult Census and Bank Marketing datasets (94%). The performance of *Fairness Contract* on the Titanic dataset is lower compared to the other datasets, primarily due to the extensive custom data preprocessing required, which limits *Fairness Contract*'s ability to detect fairness issues.

3.7.3 Efficiency (RQ2)

The results presented in Tables 3.5, 3.6, 3.7, 3.8 demonstrate that *Fairness Contract* was efficient in detecting fairness in terms of runtime. In most cases, the runtime of *Fairness Contract* is faster than the original runtime and fairness bug-fixing techniques. This efficiency is due to *Fairness Contract* detecting fairness bugs in the buggy program and aborting the program before it finishes running. Although additional time is required to verify the conditions of the contract, this extra time is still shorter compared to running the remaining parts of the ML pipeline, such as the training phase.

In some cases, such as GC2, GC8, and GC9 in Table 3.7, *Fairness Contract* has a longer runtime than other techniques because it needs to validate the individual fairness condition,

Table 3.5: Comparison of runtime between *Fairness Contract* and fairness repairing methods on Adult Census (AC)

ML Pipeline	**OR**	**R**	**DIR**	**PML**	**EO**	**FAX**	**CEO**	**ROC**	**FC**
AC1	1.41	1.72 (↑ 1.01)	7.09 (↑ 6.38)	2.23 (↑ 1.52)	1.43 (↑ 0.72)	4.03 (↑ 3.32)	1.47 (↑ 0.76)	31.07 (↑ 30.36)	0.71
AC2	22.77	35.05 (↑ 33.78)	35.81 (↑ 34.54)	24.65 (↑ 23.38)	22.80 (↑ 21.53)	40.31 (↑ 39.04)	22.82 (↑ 21.55)	45.00 (↑ 43.73)	1.27
AC3	2.59	3.68 (↑ 2.18)	6.02 (↑ 4.52)	3.12 (↑ 1.62)	2.61 (↑ 1.11)	7.89 (↑ 6.39)	2.63 (↑ 1.13)	25.50 (↑ 24.00)	1.50
AC4	2.94	7.34 (↑ 5.33)	20.38 (↑ 18.37)	4.21 (↑ 2.20)	2.98 (↑ 0.97)	8.62 (↑ 6.61)	3.03 (↑ 1.02)	41.80 (↑ 39.79)	2.01
AC5	1.27	1.64 (↑ 1.09)	7.87 (↑ 7.32)	1.51 (↑ 0.96)	1.31 (↑ 0.76)	3.63 (↑ 3.08)	1.33 (↑ 0.78)	35.68 (↑ 35.13)	0.55

↓ and ↑ indicate a longer and shorter runtime of the fairness repairing methods compared to *Fairness Contract*. OR: Original Time

Table 3.6: Comparison of runtime between *Fairness Contract* and fairness repairing methods on Bank Marketing (BM)

ML Pipeline	**OR**	**R**	**DIR**	**PML**	**EO**	**FAX**	**CEO**	**ROC**	**FC**
BM1	1.41	4.15 (↑ 3.64)	2.56 (↑ 2.07)	1.73 (↑ 1.22)	1.43 (↑ 0.92)	5.87 (↑ 5.36)	1.44 (↑ 0.93)	16.68 (↑ 16.18)	0.51
BM2	2.30	2.88 (↑ 2.25)	10.03 (↑ 9.40)	2.97 (↑ 2.34)	2.31 (↑ 1.69)	8.93 (↑ 8.30)	2.34 (↑ 1.71)	18.41 (↑ 17.78)	0.63
BM3	4.70	6.98 (↑ 5.26)	7.50 (↑ 5.77)	5.53 (↑ 3.84)	4.72 (↑ 3.00)	6.96 (↑ 5.24)	4.73 (↑ 3.01)	20.23 (↑ 18.51)	1.72
BM4	1.81	2.37 (↑ 1.61)	4.47 (↑ 3.71)	2.62 (↑ 1.86)	1.83 (↑ 1.07)	3.35 (↑ 2.59)	1.85 (↑ 1.09)	17.68 (↑ 16.92)	0.76
BM5	6.01	8.72 (↑ 6.78)	10.27 (↑ 8.33)	7.26 (↑ 5.32)	6.03 (↑ 4.10)	11.23 (↑ 9.29)	6.04 (↑ 4.11)	21.90 (↑ 19.96)	1.94
BM6	3.87	5.54 (↑ 4.62)	7.27 (↑ 6.35)	4.63 (↑ 3.71)	3.90 (↑ 2.97)	5.87 (↑ 4.95)	3.91 (↑ 2.99)	19.81 (↑ 18.87)	0.92

↓ and ↑ indicate a longer and shorter runtime of the fairness repairing methods compared to *Fairness Contract*. OR: Original Time

Table 3.7: Comparison of runtime between *Fairness Contract* and fairness repairing methods on German Credit (GC)

ML Pipeline	**OR**	**R**	**DIR**	**PML**	**EO**	**FAX**	**CEO**	**ROC**	**FC**
GC1	1.57	2.34 (↑ 1.37)	2.56 (↑ 1.56)	2.34 (↑ 1.33)	1.58 (↑ 0.57)	3.38 (↑ 2.37)	1.57 (↑ 0.56)	3.58 (↑ 2.57)	1.01
GC2	5.62	10.43 (↓ 0.20)	10.59 (↓ 0.36)	6.01 (↓ 4.22)	5.70 (↓ 4.53)	9.66 (↓ 0.57)	5.80 (↓ 4.43)	11.45 (↑ 1.22)	10.23
GC3	0.17	0.19 (↑ 0.07)	2.84 (↑ 0.16)	0.87 (↑ 0.75)	0.18 (↑ 0.06)	1.98 (↑ 1.86)	0.17 (↑ 0.05)	2.35 (↑ 2.23)	0.12
GC4	1.41	2.37 (↑ 1.02)	2.38 (↑ 1.03)	1.63 (↑ 0.28)	1.43 (↑ 0.08)	3.32 (↑ 1.97)	1.41 (↑ 0.06)	3.50 (↑ 2.15)	1.35
GC5	0.59	0.91 (↑ 0.37)	0.82 (↑ 0.28)	1.12 (↑ 0.58)	0.60 (↑ 0.06)	1.77 (↑ 1.23)	0.60 (↑ 0.05)	2.62 (↑ 2.08)	0.54
GC6	0.27	0.35 (↑ 0.00)	0.44 (↑ 0.09)	0.75 (↑ 0.40)	0.56 (↑ 0.21)	1.31 (↑ 0.96)	0.30 (↑ 0.05)	3.10 (↑ 2.75)	0.35
GC7	0.27	0.32 (↓ 0.14)	0.45 (↓ 0.01)	0.89 (↑ 0.43)	0.28 (↓ 0.18)	1.48 (↑ 1.02)	0.27 (↓ 0.19)	2.33 (↑ 1.87)	0.46
GC8	0.21	0.28 (↓ 5.39)	0.31 (↓ 5.36)	0.56 (↓ 5.11)	0.23 (↓ 5.44)	1.79 (↓ 3.88)	0.26 (↓ 5.41)	1.97 (↓ 3.70)	5.67
GC9	0.31	0.32 (↓ 6.43)	0.47 (↓ 6.30)	1.43 (↓ 5.32)	0.32 (↓ 6.43)	2.64 (↑ 4.11)	0.31 (↓ 6.44)	2.30 (↓ 4.45)	6.75

↓ and ↑ indicate a longer and shorter runtime of the fairness repairing methods compared to *Fairness Contract*. OR: Original Time

Table 3.8: Comparison of runtime between *Fairness Contract* and fairness repairing methods on Titanic (TT)

ML Pipeline	**OR**	**R**	**DIR**	**PML**	**EO**	**FAX**	**CEO**	**ROC**	**FC**
TT1	0.63	51.53 (↑ 51.20)	1.81 (↑ 1.48)	2.34 (↑ 2.01)	0.63 (↑ 0.30)	1.12 (↑ 0.79)	0.63 (↑ 0.30)	2.14 (↑ 1.81)	0.33
TT2	0.29	0.45 (↑ 0.00)	0.52 (↑ 0.07)	1.42 (↑ 0.97)	0.31 (↓ 0.14)	0.75 (↑ 0.30)	0.29 (↑ 0.38)	2.70 (↑ 2.08)	0.45
TT3	0.80	1.21 (↑ 0.79)	1.44 (↑ 1.02)	2.31 (↑ 1.89)	0.83 (↑ 0.41)	2.03 (↑ 1.61)	0.80 (↑ 0.38)	2.50 (↑ 2.01)	0.42
TT4	5.71	5.72 (↑ 4.37)	5.96 (↑ 4.60)	6.23 (↑ 4.87)	5.76 (↑ 4.40)	6.67 (↑ 5.31)	5.71 (↑ 4.36)	7.45 (↑ 6.10)	1.36
TT5	1.70	1.75 (↑ 0.79)	3.65 (↑ 2.70)	2.64 (↑ 1.68)	1.70 (↑ 0.74)	3.44 (↑ 2.48)	1.70 (↑ 0.74)	4.20 (↑ 3.23)	0.96

↓ and ↑ indicate a longer and shorter runtime of the fairness repairing methods compared to *Fairness Contract*. OR: Original Time

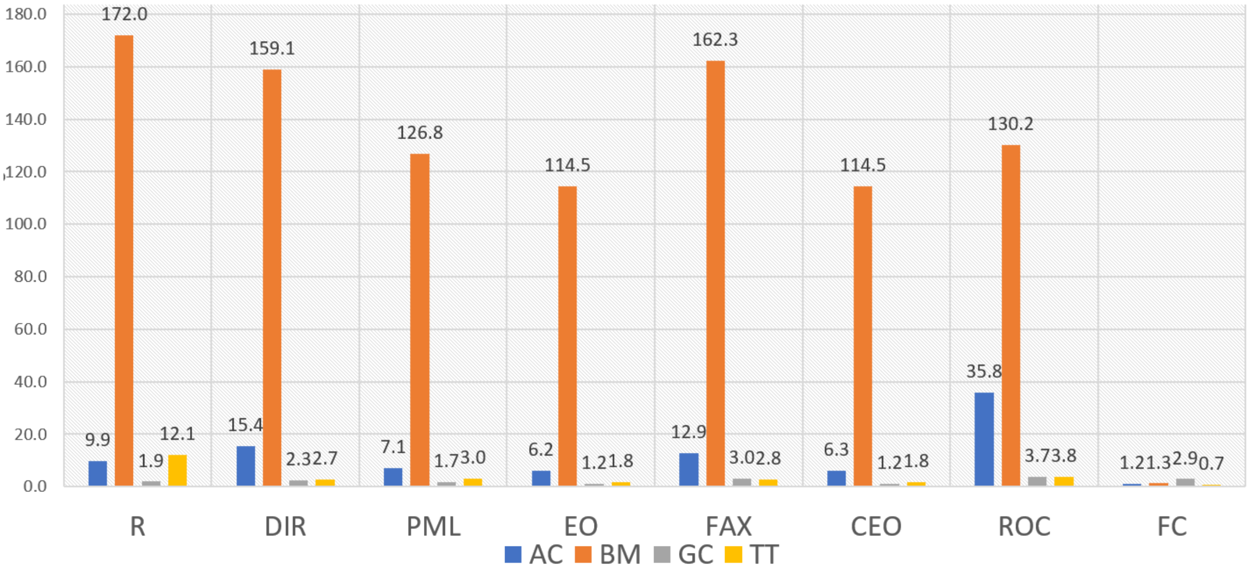


Figure 3.2: Comparison of average runtime

requiring pairwise sample checks. Despite this, *Fairness Contract*'s ability to localize fairness bugs within the ML pipeline is unmatched by other techniques. Although *Fairness Contract* can have a higher runtime in certain cases, its average runtime across datasets is significantly shorter than existing methods. As shown in 3.2, *Fairness Contract* takes only 1.2, 1.3, 2.9, and 0.7 seconds for the AC, BM, GC, and TT datasets, respectively, while the fastest other methods take 6.2, 114.5, 1.2, and 1.8 seconds for the same datasets.

We believe *Fairness Contract* is superior to current fairness testing and bug-fixing techniques in detecting and localizing fairness bugs. Existing methods often focus on a single ML pipeline component, limiting their comprehensiveness and algorithm support. Evidence in Table 3.3 shows these methods' limitations. Although we do not compare *Fairness Contract* directly with other fairness testing methods [20, 32, 64, 59], their descriptions suggest they cannot localize fairness bugs like *Fairness Contract*'s DbC approach. For instance, Aequitas [59] can generate discriminatory inputs but cannot identify their source.

3.7.4 Applicability (RQ3)

Tables 3.3 and 3.4 demonstrate the applicability of *Fairness Contract* on real-world ML programs, showing that *Fairness Contract* can detect fairness bugs in different types of ML algorithms with high precision, recall, and accuracy. This high performance is due to *Fairness Contract*'s proposed contract writing mechanism, which helps developers easily obtain fairness properties for various ML algorithms. As discussed in Section §3.6, fairness properties are diverse and challenging to obtain because most of them are probabilistic properties or hyperproperties. The *Fairness Contract* mechanism allows developers to utilize other ML libraries or write their own functions to obtain fairness properties, giving them the flexibility to write contracts and detect fairness bugs.

Fairness Contract also demonstrates its applicability across different ML datasets. It not only has high bug detection accuracy but also a short runtime on all evaluated datasets. *Fairness Contract* can detect fairness bugs quickly because it allows developers to avoid running the entire program, significantly reducing runtime. Additionally, many fairness properties proposed in our listed contracts require minimal running time. For example, to determine whether a program has a bug in splitting training and testing data (sampling), developers can check the imbalance ratio and the usage of the stratify API. If the imbalance ratio is smaller than the testing sample over the training sample and the developer does not use Stratify API, there is a high chance that the testing data will lack minority labels, causing bias in predictions. These fairness properties can be obtained in milliseconds, but they can save developers hours of time in running the entire ML pipeline and debugging the problem.

3.8 Discussion

3.8.1 Limitations

Fairness Contract has been primarily evaluated on tabular datasets and binary classification tasks using classical machine learning models. Therefore, further research is needed to assess its effectiveness, efficiency, and adaptability on other types of datasets and machine learning models. Despite this limitation, we believe the concept of using contracts to detect and localize fairness bugs is not restricted to tabular datasets and classical ML models. Additionally, some fairness properties, such as individual fairness, require high computational costs and long runtimes. However, *Fairness Contract* allows developers to design their own methods or use external libraries to extract fairness properties. This limitation can be addressed by combining *Fairness Contract* with other methods. For example, developers could combine advanced individual verification techniques with *Fairness Contract* to reduce the computational cost of verifying individual fairness.

3.8.2 Threats to Validity

Using threshold parameters for conditions in *Fairness Contract* poses several threats to validity, including false positives and false negatives. Static thresholds may not adapt to changing conditions, adding complexity to the implementation. To mitigate this threat, we complement the threshold parameters with different validation methods and datasets to ensure reliability and effectiveness. External threats may impact our proposed approach, such as imprecise precondition and postcondition definitions derived from library documentation. To mitigate this threat, we have adopted definitions from recent research studies [34, 11]

3.9 Conclusions

We present *Fairness Contract*, an innovative contract framework that aims to detect and localize fairness bugs in the ML pipeline. Our approach is extensible and generalizable, allowing for capturing fairness properties in different ML algorithms. We have designed 24 fairness contracts for a diverse range of ML algorithms, covering three main components of the ML pipeline: data preprocessing, feature engineering, and modeling. Our experiments show that *Fairness Contract* outperforms other fairness bug-fixing techniques with a higher bug detection rate and a shorter runtime. In the future, we plan to extend our approach to address additional types of fairness bugs, including those that occur in deep learning problems, beyond the scope of the current study.

Bibliography

[1] 2024. Fairness Contract. https://github.com/tess100766/FairnessContract

[2] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 625–635.

[3] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 625–635. https://doi.org/10.1145/3338906.3338937

[4] Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. 2023. Design by Contract for Deep Learning APIs. In *ESEC/FSE'2023: The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, California).

[5] Aws Albarghouthi and Samuel Vinitsky. 2019. Fairness-Aware Programming. In *Proceedings of the Conference on Fairness, Accountability, and Transparency* (Atlanta, GA, USA) *(FAT* '19)*. Association for Computing Machinery, New York, NY, USA, 211–219. https://doi.org/10.1145/3287560.3287588

[6] Edesio Alcobaça, Felipe Siqueira, Adriano Rivolli, Luís P. F. Garcia, Jefferson T. Oliva, and André C. P. L. F. de Carvalho. 2020. MFE: Towards reproducible meta-feature extraction. *Journal of Machine Learning Research* 21, 111 (2020), 1–5. http://jmlr.org/papers/v21/19-348.html

[7] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2018. Themis: automatically testing software for discrimination. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 871–875. https://doi.org/10.1145/3236024.3264590

[8] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine bias risk assessments in criminal sentencing. *ProPublica, May* 23 (2016).

[9] Rachel KE Bellamy, Kuntal Dey, Michael Hind, Samuel C Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, et al. 2018. AI Fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *arXiv preprint arXiv:1810.01943* (2018).

[10] Sumon Biswas and Hridesh Rajan. 2020. Do the Machine Learning Models on a Crowd Sourced Platform Exhibit Bias? An Empirical Study on Model Fairness. In *ESEC/FSE'2020: The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States).

[11] Sumon Biswas and Hridesh Rajan. 2021. Fair preprocessing: towards understanding compositional fairness of data transformers in machine learning pipeline. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 981–993. https://doi.org/10.1145/3468264.3468536

[12] Sumon Biswas and Hridesh Rajan. 2023. Fairify: Fairness Verification of Neural Networks. In *ICSE'2023: The 45th International Conference on Software Engineering* (Melbourne, Australia).

[13] Miranda Bogen and Aaron Rieke. 2018. Help wanted: an examination of hiring algorithms, equity, and bias. https://api.semanticscholar.org/CorpusID:158203520

[14] Ajay Byanjankar, Markku Heikkilä, and Jozsef Mezei. 2015. Predicting credit risk in peer-to-peer lending: A neural network approach. In *2015 IEEE symposium series on computational intelligence*. IEEE, 719–725.

[15] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data mining and knowledge discovery* 21, 2 (2010), 277–292.

[16] L Elisa Celis, Lingxiao Huang, Vijay Keswani, and Nisheeth K Vishnoi. 2019. Classification with fairness constraints: A meta-algorithm with provable guarantees. In *Proceedings of the conference on fairness, accountability, and transparency*. 319–328.

[17] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in machine learning software: Why? how? what to do?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 429–440.

[18] Joymallya Chakraborty, Suvodeep Majumder, Zhe Yu, and Tim Menzies. 2020. Fairway: a way to build fair ML software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 654–665.

[19] Joymallya Chakraborty, Kewen Peng, and Tim Menzies. 2021. Making fair ML software using trustworthy explanation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1229–1233. https://doi.org/10.1145/3324884.3418932

[20] Zhenpeng Chen, Jie M. Zhang, Max Hort, Mark Harman, and Federica Sarro. 2024. Fairness Testing: A Comprehensive Survey and Analysis of Trends. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 137 (jun 2024), 59 pages. https://doi.org/10.1145/3652155

[21] Zhenpeng Chen, Jie M Zhang, Federica Sarro, and Mark Harman. 2022. MAAT: a novel ensemble approach to addressing fairness and performance bugs for machine learning software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1122–1134.

[22] Alexandra Chouldechova. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data* 5, 2 (2017), 153–163.

[23] Alexandra Chouldechova, Diana Benavides-Prado, Oleksandr Fialko, and Rhema Vaithianathan. 2018. A case study of algorithm-assisted decision making in child maltreatment hotline screening decisions. In *Conference on Fairness, Accountability and Transparency.* PMLR, 134–148.

[24] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (sep 2010), 1157–1210.

[25] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing* (2009), 1–4.

[26] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O'Donoghue, Daniel Visentin, et al. 2018. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine* 24, 9 (2018), 1342–1350.

[27] Virginia Eubanks. 2018. *Automating inequality: How high-tech tools profile, police, and punish the poor.* St. Martin's Press.

[28] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining.* 259–268.

[29] Anthony Finkelstein, Mark Harman, S. Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. 2008. “Fairness Analysis” in Requirements Assignments. In *2008 16th IEEE International Requirements Engineering Conference*. 115–124. https://doi.org/10.1109/RE.2008.61

[30] Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. 2019. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the conference on fairness, accountability, and transparency*. 329–338.

[31] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. 498–510.

[32] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 498–510. https://doi.org/10.1145/3106237.3106277

[33] Xuanqi Gao, Juan Zhai, Shiqing Ma, Chao Shen, Yufei Chen, and Qian Wang. 2022. FairNeuron: improving deep neural network fairness with adversary games on selective neurons. In *Proceedings of the 44th International Conference on Software Engineering*. 921–933.

[34] Usman Gohar, Sumon Biswas, and Hridesh Rajan. 2023. Towards Understanding Fairness and its Composition in Ensemble Machine Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE ’23)*. IEEE Press, 1533–1545. https://doi.org/10.1109/ICSE48619.2023.00133

[35] Przemyslaw A Grabowicz, Nicholas Perello, and Aarshee Mishra. 2022. Marrying fairness and explainability in supervised learning. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. 1905–1916.

[36] Brett Graham, William Furr, Karol Kuczmarski, Bernhard Biskup, and Adam Palay. 2010. *PyContracts*. https://andreacensi.github.io/contracts//

[37] Moritz Hardt, Eric Price, and Nati Srebro. 2016. Equality of opportunity in supervised learning. *Advances in neural information processing systems* 29 (2016).

[38] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the eighth international workshop on data mining for online advertising*. 1–9.

[39] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

[40] Mitchell Hoffman, Lisa B Kahn, and Danielle Li. 2018. Discretion in hiring. *The Quarterly Journal of Economics* 133, 2 (2018), 765–800.

[41] Max Hort and Federica Sarro. 2021. Did you do your homework? Raising awareness on software fairness and discrimination. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1322–1326.

[42] Max Hort, Jie M Zhang, Federica Sarro, and Mark Harman. 2021. Fairea: A model behaviour mutation approach to benchmarking bias mitigation methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 994–1006.

[43] Kaggle. 2017. Adult Census Dataset. https://www.kaggle.com/datasets/uciml/adult-census-income

[44] Kaggle. 2017. Bank Marketing Dataset. https://www.kaggle.com/c/bank-marketing-uci

[45] Kaggle. 2017. German Credit Dataset. https://www.kaggle.com/datasets/uciml/german-credit

[46] Kaggle. 2017. Titanic ML Dataset. https://www.kaggle.com/c/titanic/data

[47] Faisal Kamiran and Toon Calders. 2012. Data preprocessing techniques for classification without discrimination. *Knowledge and information systems* 33, 1 (2012), 1–33.

[48] Faisal Kamiran, Asim Karim, and Xiangliang Zhang. 2012. Decision theory for discrimination-aware classification. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 924–929.

[49] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. 2012. Fairness-aware classifier with prejudice remover regularizer. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 35–50.

[50] Konstantina Kourou, Themis P Exarchos, Konstantinos P Exarchos, Michalis V Karamouzis, and Dimitrios I Fotiadis. 2015. Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal* 13 (2015), 8–17.

[51] Oliver Kramer and Oliver Kramer. 2016. Scikit-learn. https://scikit-learn.org/

[52] Yanhui Li, Linghan Meng, Lin Chen, Li Yu, Di Wu, Yuming Zhou, and Baowen Xu. 2022. Training data debugging for the fairness of machine learning software. In *Proceedings of the 44th International Conference on Software Engineering*. 2215–2227.

[53] Milad Malekipirbazari and Vural Aksakalli. 2015. Risk assessment in social lending via random forests. *Expert Systems with Applications* 42, 10 (2015), 4621–4631.

[54] B. Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[55] Giang Nguyen, Sumon Biswas, and Hridesh Rajan. 2023. Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software*

Engineering (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/3611643.3616257

[56] Luca Oneto, Michele Donini, Andreas Maurer, and Massimiliano Pontil. 2019. Learning fair and transferable representations. *arXiv preprint arXiv:1906.10673* (2019).

[57] Dino Pedreshi, Salvatore Ruggieri, and Franco Turini. 2008. Discrimination-aware data mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 560–568. https://doi.org/10.1145/1401890.1401959

[58] Claudia Perlich, Brian Dalessandro, Troy Raeder, Ori Stitelman, and Foster Provost. 2014. Machine learning for targeted display advertising: Transfer learning in action. *Machine learning* 95, 1 (2014), 103–127.

[59] Pedro Saleiro, Benedict Kuester, Loren Hinkson, Jesse London, Abby Stevens, Ari Anisfeld, Kit T Rodolfa, and Rayid Ghani. 2018. Aequitas: A bias and fairness audit toolkit. *arXiv preprint arXiv:1811.05577* (2018).

[60] Guanhong Tao, Weisong Sun, Tingxu Han, Chunrong Fang, and Xiangyu Zhang. 2022. RULER: discriminative and iterative adversarial training for deep neural network fairness. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1173–1184.

[61] Saeid Tizpaz-Niari, Ashish Kumar, Gang Tan, and Ashutosh Trivedi. 2022. Fairness-aware Configuration of Machine Learning Libraries. *arXiv preprint arXiv:2202.06196* (2022).

[62] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 401–416.

[63] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 98–108.

[64] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 98–108. https://doi.org/10.1145/3238147.3238165

[65] Rhema Vaithianathan, Tim Maloney, Emily Putnam-Hornstein, and Nan Jiang. 2013. Children in the public benefit system at risk of maltreatment: Identification via predictive modeling. *American journal of preventive medicine* 45, 3 (2013), 354–359.

[66] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rogriguez, and Krishna P Gummadi. 2017. Fairness constraints: Mechanisms for fair classification. In *Artificial Intelligence and Statistics*. PMLR, 962–970.

[67] Brian Hu Zhang, Blake Lemoine, and Margaret Mitchell. 2018. Mitigating unwanted biases with adversarial learning. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*. 335–340.

[68] Lingfeng Zhang, Yueling Zhang, and Min Zhang. 2021. Efficient white-box fairness testing through gradient search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/3460319.3464820

[69] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 949–960.

CHAPTER 4. MODULAR FAIRNESS CHECKER FOR MACHINE LEARNING PIPELINE

Giang Nguyen[1], Shibbir Ahmed[2], Sumon Biwas[3], and Hridesh Rajan[4]

[1] Iowa State University, USA

[2] Texas State University, USA

[3] Carnegie Mellon University, USA

[4] Tulane University, USA

Modified from a manuscript submitted in *The proceedings of the 33th ACM Joint Meeting on European Software Engineering (ESEC/FSE '25)*

4.1 Abstract

Machine learning (ML) systems are increasingly used in high-stakes decision-making processes, raising concerns about fairness and potential biases. Addressing these concerns requires new tools and methods to detect bias in ML-based software effectively. Current bias detection techniques typically analyze *global fairness* that evaluate the fairness of the ML pipeline holistically. However, this approach has limitations in identifying the pipeline component responsible for bias and explaining the root cause. In addition, developers lack methods to apply modular fairness constraints in design time. We propose a novel approach that leverages concentration inequalities to detect bias in the ML pipeline, focusing on *modular fairness*. Our method begins by computing fairness scores for the dataset features, highlighting which features are positively or negatively impacted after applying preprocessing components in the pipeline. These insights are then applied to concentration inequalities to detect the fairness bugs within the ML pipeline. Additionally, we developed a programming language abstraction that allows developers to specify and enforce modular fairness during ML pipeline development. We evaluated our approach using four datasets,

two fairness metrics, and 13 ML algorithms. *Fairness Checker* has a high success rate in detecting fairness bugs. Specifically, our method identified 42 out of 45 buggy cases with an average processing time of 26 seconds per case, outperforming existing bias detection methods, which identified up to 35 out of 45 cases with an average of 31 seconds per case.

4.2 Introduction

Recent advancements in machine learning have achieved significant success in addressing complex decision-making challenges, including job recommendations, employee hiring, social services, and education [21, 20, 45, 59, 44, 65, 29, 71, 30, 56, 31, 63, 10]. In these contexts, ensuring that decisions are made fairly and without bias is crucial for both ethical and legal reasons. One of the key motivations for incorporating machine learning in these scenarios is the belief that machines can avoid the implicit biases that often influence human decision-makers. However, designing ML pipelines that meet fairness standards has proven to be quite challenging due to their inherent complexity since various components, including data preprocessing and feature engineering, significantly contribute to the overall fairness of the system.

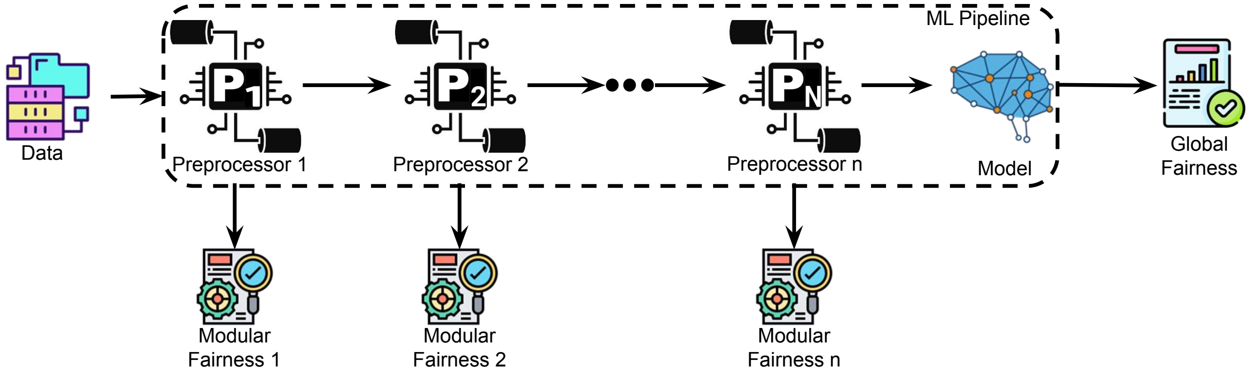


Figure 4.1: ML Pipeline with data preprocessing modules, each having individual fairness impact.

Concerns about fairness in software have been explored by both the Software Engineering (SE) [34] and Machine Learning (ML) [64] research communities. The SE community has concentrated on developing testing and verification methods aimed at detecting unfairness within software systems [18, 36, 9, 37, 69, 25, 38, 70, 13]. The ML research community has extensively studied fairness by developing various metrics and mitigation methods [22, 28, 33, 42, 53, 55, 72, 73]. A variety of bias mitigation techniques have been developed, including altering training data to remove bias (pre-processing) [58, 24, 23], modifying the model building process itself (in-processing) [66, 27, 47, 39, 67], or adjusting the final results to ensure fairness (post-processing) [9, 69, 75].

Biswas *et al.* proposed *Fair Preprocessing* [17], which demonstrated that different ML pipeline components have their own *modular fairness* that significantly impact the *global fairness* of ML pipelines [40, 62], illustrated in Figure 4.1. *Modular fairness* refers to the fairness value calculated for individual components (or modules) within an ML pipeline. *Global fairness* value refers to a holistic measure of fairness calculated based on the overall output or performance of the entire ML pipeline. So, If these components are not applied correctly, they may inadvertently increase bias in the final predictions of the ML system. For instance, the *Min-Max Scaler* algorithm may introduce bias when used in conjunction with the *K-Neighbors* classifier, yet it remains fair when paired with the *XGBoost* classifier. Unfortunately, existing bias mitigation and testing techniques are not able to detect such context-specific fairness issues [46, 25]. These bias testing methods focus on identifying fairness issues in ML software by analyzing *global fairness*. They might detect individual fairness violations by examining the dataset after it has been processed by the entire ML pipeline (*global fairness* analyzing). Similarly, bias mitigation techniques still have limitations in localizing the source of the bias, as they primarily focus on adjusting the *global fairness* of the ML pipeline rather than identifying the specific origins of bias.

To ensure high fairness for the ML pipeline, the developer needs to ensure each component in ML does not introduce new biases to the ML pipeline. By focusing on problems related to *modular fairness*, the developer can detect and address biases early in the ML pipeline [17]. If biases are identified at the data preprocessing or feature selection stages, they can be corrected before propagating through the rest of the pipeline, leading to better *global fairness*. Moreover, when fairness is analyzed at the modular level, localizing the source of bias becomes easier. If a particular module, such as feature engineering or model training, is found to introduce bias, it can be modified or replaced without overhauling the entire system. However, it is challenging to detect and explain the bias problem inside each preprocessor of the pipeline. ML pipelines consist of multiple interconnected modules (e.g., data preprocessing, feature selection, model training, and decision-making) [19]. Each module can influence the fairness of subsequent modules, creating complex dependencies. A bias introduced in one module can propagate or be amplified by others, making it difficult to isolate the source of unfairness and understand its full impact. Fairness can be defined in various ways, and different models or systems may require distinct fairness metrics to ensure appropriate evaluation. A metric appropriate for fairness in feature selection might not be suitable for assessing fairness in model training. Choosing the right metric and ensuring consistency across modules is challenging.

We propose *Fairness Checker*, which **detects** fairness bugs at the modular level and **explains** their root causes. Unlike existing bias mitigation and testing techniques, our approach not only identifies fairness bugs but also addresses interdependencies in ML components and varying fairness metrics. Unlike current bias explanation methods [17, 40, 46] that only do empirical analysis, we introduce an automated tool leveraging concentration inequalities for strong soundness guarantees. Instead of analyzing *global fairness*, we observe *modular fairness* to identify fairness bugs. Our method includes fairness annotators, allowing users to specify whether a preprocessor enhances fairness, as shown in Example 4.1. In this example, a developer specifies whether applying a preprocessor should increase the *parity difference* (SPD) by more than 0.2 by using annotation *@fairness_contract*. By invoking *diff*, our novel mutation feature importance approach estimates modular feature scores using returned value *X_trans* and *y_trans*, representing the feature scores of the data after the preprocessor is applied. These scores help developers identify which data features are negatively impacted by the preprocessor. Additionally, these feature scores are used in concentration inequalities to check whether the fairness constraints that are defined by the user. For instance, the system might be configured to report a violation only if there is an 85% probability that the fairness specification has been breached.

```
@fairness_contract(diff((X, y), (X_trans, y_trans), 'SPD', 'bagging') ≥ 0.2)
def preprocessor(X, y):
    ...
    return X_trans, y_trans
```

Listing 4.1: Modular Fairness Specification Example

Together, this information provides developers with a comprehensive understanding of bias issues within their ML pipeline by analyzing feature scores and violation reports, enabling them to make more informed decisions about fairness improvements. First, the developer can be warned when the module violates fairness, as shown in Example 4.2. Second, the developer can enforce modular constraints to ensure the data passes through a fair preprocessor before reaching the classifier.

```
"Violation Message: This algorithm cannot decrease the SPD up to 0.2 and negatively affect
    the features 1 and 2 of the data."
```

Listing 4.2: Fairness Violation Message for Example 4.1

Fairness Checker is designed to overcome the limitations of existing bias detection techniques by leveraging concentration inequalities to efficiently detect and explain modular fairness bugs across diverse datasets, fairness metrics, and ML algorithms. We conducted an extensive evaluation of *Fairness Checker* using four widely recognized datasets from the fairness literature [37, 69, 9] and 45 buggy cases from a recent study [17]. The results highlight the effectiveness of our approach: *Fairness Checker* successfully identified 42 out of 45 buggy cases, with an average processing time of 26 seconds per case, surpassing existing bias detection methods, which detected up to 35 cases with an average time of 31 seconds.

Our main contributions are the following:

- We designed fairness annotators to specify fairness properties during implementation time.
- We propose a novel fairness-aware feature importance technique to explain *modular fairness*.
- We propose a novel method to detect *modular fairness* bugs using concentration inequalities.
- We designed *Fairness Checker* [8] that is extensible to diverse ML algorithm types.

Table 4.1: Local fairness of stages as downstream transformer. [17]

Model	Stage	SF_SPD	SF_EOD	SF_AOD	SF_ERD
XGB	MA	-0.036	0.005	-0.059	-0.128
KNC	MA	0.079	0.114	0.075	-0.078

The paper is organized as follows: §4.3 presents a motivation, §4.4 describes the background, §4.5 discusses the problem formula, §4.6 shows the *Fairness Checker* approaches, §4.7 presents our evaluation, §4.8 discusses the threats to validity, §4.9 concludes.

4.3 Motivation

Biswas *et al.* propose *Fair Preprocessing* [17], which demonstrated that different ML preprocessing algorithms can significantly impact the fairness of an ML pipeline. These algorithms must be applied correctly to avoid inadvertently increasing bias in the final predictions of the ML system. Table 4.1 highlights that various preprocessing algorithms affect different classifiers differently. This table also shows that each ML pipeline component has its own *modular fairness*. For instance, the *Max-Abs* (MA) scaler reduces bias regarding SPD and AOD for the *XGBoost* classifier but introduces more bias in these metrics for the *K-Neighbors* classifier. This result underscores the importance of selecting appropriate preprocessing algorithms for each classifier. However, this study only provides an empirical analysis of how different preprocessors affect the overall outcome without offering automated methods to identify the issues.

Moreover, various bias mitigation techniques have been developed to enhance the fairness of ML systems. Additionally, several fairness specification methods have been introduced to check and verify whether the ML system meets a predefined fairness requirement. However, these techniques also have their limitations. Although these methods can improve, test, or verify fairness, they fall short of explaining or pinpointing the root causes of bias. Additionally, they are limited to detecting fairness issues within a specific machine learning model and lack the ability to generalize across a broader range of machine learning algorithms [32, 33, 42, 26]. To illustrate the limitation of these methods, we have evaluated whether existing bias detection techniques, including *Reweighing* [53],

Table 4.2: Average proportion of accurately detected bugs by fairness bug detection techniques.

	R	DIR	FAX	CEO	ROC	**Mean**
Sampling (15)	13%	33%	27%	67%	33%	**34.6%**
FN (6)	33%	17%	0%	17%	33%	**20.0%**
FS (11)	27%	18%	37%	27%	27%	**27.2%**

Disparate Impact Remover [33], *FaX-AI* [41], *Reject Option Classification* [54], can identify fairness issues caused by components of the ML pipeline. We assessed 32 buggy cases, each caused by different components of the ML pipeline, including sampling, feature normalization (FN), and feature selection (FS). These cases were identified by the *Fair Preprocessing* study as introducing new biases when specific preprocessing methods were applied to the pipeline. The results, presented in Table 4.2, show the percentage of each technique's ability to detect bias in these cases.

Table 4.2 illustrates that the majority of existing bias detection techniques have a poor percentage of detecting fairness bugs caused by a specific ML pipeline component. Specifically, these techniques detected only 34.6% of the buggy cases introduced by sampling algorithms, 20% of the buggy cases caused by feature normalization algorithms, and 27.2% of the buggy cases resulting from feature selection algorithms. The main reason for these results is that not all the modular fairness bugs significantly reduce the fairness of the whole ML pipeline; therefore, it is challenging to detect by existing bias detection techniques. However, since these modular fairness bugs also reduce fairness to the entire ML pipeline, we can increase the fairness of the pipeline if we can detect the bug and fix it.

Inspired by *Fair Preprocessing*, we propose a novel method to automatically detect and explain an ML pipeline's modular fairness, addressing the limitations of the existing approaches. Example 4.1 illustrates how our approach can help developers detect and explain modular fairness in practice. The detailed approach is discussed in Section §4.6.

4.4 Background

4.4.1 Concentration Inequalities

Concentration inequalities provide bounds on how a random variable deviates from some expected value, typically with high probability. In fairness problems, they are crucial for understanding the performance and fairness of algorithms by quantifying the uncertainty or deviation of statistical estimates or model predictions [11, 14]. For example, McDiarmid's inequality estimates the range of bias value change after a preprocessing method. McDiarmid's inequality applies to a function $f(x_1, ..., x_n)$ that depends on several independent random variables $x_1, ..., x_n$. This idea is formalized through the concept of bounded differences. We have McDiarmid's inequality defined as follows:

$$Pr[|f(x_1, \ldots, x_j, \ldots, x_n) - \mathbb{E}[f(x_1, \ldots, x_j, \ldots, x_n)]| \geq \epsilon] \leq exp(\frac{-2\epsilon^2}{\sum_{j=1}^{n} c_j^2}) \tag{4.1}$$

Where c_j is a sensitivity that bounds how much f can change when only p_i is varied while the others are held constant.

4.4.2 Sensitivity

In the context of concentration inequalities, sensitivity refers to how much a function's output can change when its input is altered by a small amount, usually by changing one element of a vector or a sample. It provides a measure of how sensitive the function is to changes in its input. More formally, the sensitivity of this function can be defined as the maximum change in f's value when one of the input elements is changed while keeping all others fixed. the sensitivity with respect to the j^{th} variable is given by:

$$c_j = \sup_{j \in M} |f(x_1, \ldots, x_j, \ldots, x_n) - f(x_1, \ldots, x'_j, \ldots, x_n)| \tag{4.2}$$

Here, x'_j represents a possible change in the j^{th} element of the input. sup refers to the supremum of a set, which is the least upper bound of that set.

4.4.3 Related Work

4.4.3.1 Fairness Specification

Recent research has focused on specifying fairness in machine learning software [11, 14, 74, 60]. For example, Albarghouthi et al. introduced fairness-aware programming [11], enabling declarative fairness definitions within the decision-making code specified at runtime. Similarly, OmniFair [74] allows developers to specify group fairness constraints in machine learning, supporting all major group fairness constraints, including custom ones, while handling multiple constraints concurrently. However, these methods struggle to identify specific fairness issues and demonstrate their applicability across various ML algorithms. Like these methods, *Fairness Checker* uses fairness annotators for fairness specification and applies concentration inequalities to check fairness. However, our method specifies fairness during implementation, not at runtime, and focuses on modular fairness to detect and explain bias, improving fairness bug detection and resolution.

Design by contract methodology. These methods [11, 74] allow users to write fairness annotations that can be used to specify fairness constraints or conditions during runtime. This means that fairness checks are applied while the program executes, allowing the system to evaluate whether specific fairness criteria are met at different stages of the model's operation. In contrast, *Fairness Checker* takes a proactive approach by allowing developers to specify fairness constraints during the implementation phase before the model or system is even run. By embedding fairness annotations at the design and development stages, *Fairness Checker* ensures that fairness is considered and enforced earlier in the machine learning pipeline. This shift to implementation-time fairness annotation allows developers to catch potential biases or fairness bugs before they become part of the operational system.

Concentration inequalities. These methods [11, 74] typically rely on Hoeffding's inequality to enforce fairness properties at runtime. In contrast, our approach leverages McDiarmid's inequality to define fairness properties during the implementation phase before the model is executed. McDiarmid's inequality offers a key advantage by providing tighter bounds on how changes in individual inputs influence the overall output, making it particularly effective for measuring sensitivity to fairness concerns. In addition, we introduce a novel method for calculating this sensitivity, specifically tailored to capture the characteristics of machine learning preprocessors and their compatibility with other preprocessing techniques. This enhancement strengthens *Fairness Checker*, making it a powerful tool for detecting fairness issues during implementation.

4.5 Problem Formulation

In this section, we formulate the problem that *Fairness Checker* aims to solve.

4.5.1 Modular Fairness Checker Inputs

4.5.1.1 ML Pipeline

Our objective is to introduce a programming language abstraction for specifying fairness in ML pipeline development. This approach allows developers to explicitly declare fairness requirements modularly for their pipeline P: $X \rightarrow \hat{y}$, which maps an input $x \in X$ (e.g., a person's demographic) to a single binary output $\hat{y} = \{0, 1\}$ (e.g., loan approval). The pipeline P is composed of preprocessing components $p_1, \ldots, p_n$, followed by a model m. Adopting this method enables a more structured and transparent articulation of fairness criteria within the development process [11, 14].

4.5.1.2 Base Models

To detect fairness bugs during the implementation process where the model m has not been finalized. Different base models provide diverse perspectives of the impact. For example, distance-based models such as the *K-Neighbors* classifier require data to be normalized, whereas tree-based models are generally not affected by the scale of the data [17]. Thus, we use base models to represent different types of machine learning models [57]. These include linear models, nearest neighbors, support vector machines, decision trees, bagging, and boosting methods. Specifically, we represent these models using *Logistic Regression* [6] for linear models, *K Neighbors Classifier* [4] for nearest neighbors, *Linear SVC* [5] for support vector machines, *Decision Tree Classifier* [1] for decision trees, *Random Forest Classifier* [7] for bagging, and *Gradient Boosting Classifier* [3] for boosting.

These base models utilize the default hyperparameters from the *Scikit-learn* library [57]. The choice of base models is intentional, as they are simple and can be trained quickly on the dataset. Their primary purpose is to highlight the strengths and weaknesses of the actual model, enabling us to estimate the impact of the preprocessing step on the features without incurring the high computational costs associated with the actual model. Our experiments and related research [57, 17, 40] demonstrate that different model types are only compatible with specific preprocessing algorithms and the characteristics of the dataset. Therefore, rather than using a complex original model, we believe employing a simpler representative model is sufficient to identify fairness issues during the implementation process [40, 57, 17]. For instance, Fair-AutoML [62] also leverages a base model to reduce AutoML's search space when addressing fairness issues. Please note that all equations following this point will use the base model m to represent the actual model.

4.5.2 Modular Fairness Checker Definition

Suppose, we have real-world data D_e on which a deployed model needs to make predictions. Our goal is to modularly verify a preprocessor p_i on this data to determine whether it introduces any bias. Formally, given a preprocessor $p_i : D \rightarrow D'$, a specification S, and a confidence level λ, our goal is to determine whether S is true with probability at least δ. Let P and P_i represent the pipeline configurations with and without the preprocessor p_i, respectively. To determine if a fairness bug is caused by p_i, we aim to compute the following measure of difference, Δ_i:

$$\Delta_i = P_i(D_e) - P(D_e) \tag{4.3}$$

If $\Delta_i < 0$, then p_i introduces bias into the pipeline. Conversely, if $\Delta_i > 0$, then p_i increases the fairness of the pipeline. Estimating *modular fairness* of p_i during the implementation is challenging:

- Evaluation data D_e might be not available.

- The training model might not be readily available for developers
- Omitting the preprocessor p_i could disrupt the pipeline, making Δ_i either unobtainable or inaccurate.

Thus, to check the fairness bug of p_i during the implementation, we estimate the probability that Δ_i exceeds a user-defined threshold ϵ using the training data D_t instead of evaluation data D_e. Furthermore, unfairness primarily arises from issues within the training data [62]. Therefore, this approach also functions as a form of training data debugging by applying modular constraints:

$$Pr[|P_i(D_t) - P(D_t)| \geq \epsilon] \leq \delta \tag{4.4}$$

By estimating δ, we can verify that the fairness of p_i exceeds a predefined threshold with high probability. The Sections §4.6 provides a detailed explanation of how we address the above challenges to specify modular fairness, using Equation 4.4.

4.5.3 Modular Fairness Specification

Next, we describe how fairness specifications from the machine learning literature [15, 36, 16] can be formalized in our specification language; the best fairness specification to use is often context-specific. We discuss additional specifications that can be represented in our language in Section §4.6. We are interested in ensuring fairness between two groups, A and B, which could represent different demographic groups (e.g., male vs. female or different racial groups). Let consider $p_A = Pr(\hat{y} = 1|z = A)$ be the favorable label's probability for the unprivileged group and $p_B = Pr(\hat{y} = 1|z = B)$ be the favorable label's probability for the privileged group. Given two different ML pipelines P and P', the **demographic parity** (SPD) [33, 72] properties are:

$$Y_{parity} \equiv (((p_A - p_B) - (p'_A - p'_B)) \geq \epsilon) \tag{4.5}$$

Let consider $t_A = Pr[\hat{y} = 1|y = 1, z = A]$ be the true-positive rate of the unprivileged group and $t_B = Pr[\hat{y} = 1|y = 1, z = B]$ be the true-positive rate of the privileged group. Given two different ML pipelines P and P', Then, the **equal opportunity** (EOD) [43] properties are:

$$Y_{equal} \equiv (((t_A - t_B) - (t'_A - t'_B)) \geq \epsilon) \tag{4.6}$$

Consider a hiring example with a male subpopulation (p_A or t_A) and a female subpopulation (p_B or t_B). The decision pipelines, P and P', determine whether a candidate, based on their years of experience and college ranking, is offered a job. The specification says that for every male candidate offered a job, at least ϵ more female candidates should be offered a job under pipeline P compared to pipeline P'.

4.6 Modular Fairness Checker

In this section, we present our approach to detecting modular fairness bugs. We introduce our modular fairness specification syntax, explain how concentration inequalities help developers specify and detect fairness bugs, and introduce a fairness-aware feature importance technique. This technique estimates feature scores, serving as a sensitivity measure for concentration inequalities and identifying which features are negatively impacted by the ML algorithm.

4.6.1 Modular Fairness Specification Syntax

The syntax and semantics of the specifications we aim to check are illustrated in Figure 4.2. In this grammar, the start symbol is $\langle S \rangle$, and the rest of the grammar allows us to build arithmetic expressions represented by $\langle E \rangle$. Essentially, this specification language lets us encode arithmetic relationships among different conditional expectations that are expected to hold. The benefit of introducing this specification language is its flexibility, enabling various fairness specifications within the same framework.

$$
\begin{aligned}
\langle T \rangle \; ::= \;& \mathbb{E}[\langle E \rangle] \\
\mid \;& \mathbb{E}[\langle E \rangle | \langle E \rangle] \\
\mid \;& c \in \mathbb{R} \\
\mid \;& \langle T \rangle \; \{+, -, \div, \times\} \; \langle T \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle S \rangle \; ::= \;& \langle T \rangle \; \{=, \neq, \leq, \geq\} \; c \\
\mid \;& \langle S \rangle \wedge \langle S \rangle \\
\mid \;& \langle S \rangle \vee \langle S \rangle
\end{aligned}
$$

Figure 4.2: Specification syntax. Here, S and T are nonterminal symbols, with S serving as the start symbol. The symbol $\langle E \rangle$ represents expressions of random variables.

An expression, denoted as $\langle E \rangle$, is a mathematical expression that depends on two variables: the input variable x and another variable r, where r represents the value of the function $f(x)$. In this context, the expectation $\mathbb{E}[r > 0]$ is used to indicate the probability that the function $f(x)$ produces a positive result. It's important to observe that both conditional probabilities and expectations can be represented within the framework of the grammar being used. For example, a conditional probability such as $P[E_1|E_2]$ (which reads as the probability of event E_1 occurring given that E_2 has occurred) can be expressed in terms of expectations. Specifically, this conditional probability can be written as the ratio of two expectations:

$$P[E_1|E_2] = \frac{\mathbb{E}[E_1 \wedge E_2]}{\mathbb{E}[E_2]}.$$

Here, $\mathbb{E}[E_1 \wedge E_2]$ represents the expected value of the conjunction of the two events (essentially the joint probability of both events happening), and $\mathbb{E}[E_2]$ is the expected value corresponding to event E_2, which acts as the normalizing factor for the conditional probability.

To illustrate the problem addressed in this work, consider a scenario where an ML developer is building a pipeline, P, to select job applicants for interviews. Using existing fairness techniques, the developer wants to ensure that the pipeline meets group fairness standards, with similar selection rates for minority and majority groups. However, the developer recognizes that this fairness may not hold after deployment, especially if the preprocessor is unsuitable or incompatible with other components. To mitigate this, the developer annotates the procedure with a *preprocessor* as follows:

```
@fairness_contract(diff((X, y), (X_trans, y_trans), 'SPD', 'boosting') ≥ 0.2)
def scaler(X, y):
    X_trans, y_trans = scaler.fit_transform(X, y)
    return X_trans, y_trans
```

Listing 4.3: Checking group fairness for a preprocessing module

Once the *preprocessor* is implemented and has transformed the data, the fairness annotator will automatically check the *preprocessor* by comparing the input data with the transformed data. This checking process occurs in the background using concentration inequalities. The developer must specify the type of fairness metric (e.g., SPD or EOD) and the model (e.g., boosting or bagging) they intend to use for the ML pipeline. The developer can set the desired level of confidence for reporting fairness violations. For example, the developer might configure the system to report a violation only if there is an 85% probability that the fairness specification has been breached.

In addition to detecting modular fairness for group fairness, we have also proposed other modular fairness specifications that are integrated into our method, focusing on probabilistic properties. These specifications aim to capture various characteristics of a *preprocessor*, such as imbalance ratio, skewness, and more. These characteristics can be calculated by assessing the probability distribution of different features within the data [12]. This specification is performed automatically in the background using the *Pr(.)* function. For instance, to determine whether there is an issue with how a program splits training and testing data (sampling), developers can evaluate the imbalance ratio and the use of the *stratify* API. If the imbalance ratio is smaller in the testing sample compared to the training sample and the developer has not used the *stratify* API, there is a high likelihood that the testing data will lack minority labels, potentially leading to bias in predictions. This specification can be applied to assess various probabilistic fairness properties.

```
@fairness_contract(Pr(y) / Pr(¬y) > 4)
def preprocessor(X, y):
    X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
    return  X_train, X_test, y_train, y_test
```

Listing 4.4: Checking imbalance ratio for the preprocessor

Our method also specifies fairness hyperproperties [11], such as individual fairness. For example, selects a representative data subset to reduce the computational load or balance class distributions, but it can introduce bias, especially with upsampling techniques [17, 23]. Checking the label ratio is essential before applying sampling. While upsampling addresses imbalance, it can create unwanted samples, causing individual bias. Thus, developers should check individual fairness after using upsampling algorithms. To specify fairness hyperproperties, the developer can call the *sim(.)* function, which counts the number of individual fairness violations in both the original and transformed datasets to assess modular individual fairness. This *sim(.)* function is implemented in the background, automatically checking whether the transformed data exhibits more individual fairness violations compared to the original data.

```
@fairness_contract(sim((X, y), (X_trans, y_trans))
def preprocessor(X, y):
    sm = SMOTE(random_state=42)
    X_trans, y_trans = sm.fit_resample(X, y)
    return X_trans, y_trans
```

Listing 4.5: Checking individual fairness for the preprocessor

4.6.2 Modular Fairness Checking using Concentration Inequality

We have that pipeline P_i has a missing component compared to the original pipeline P, so there will be some error ϵ between the estimate $P_i(D_t)$ and the true expected value $P(D_t)$. Our algorithm uses concentration inequalities to establish high probability bounds for this error. These bounds are then used to determine, with high probability, whether the fairness specification holds. By applying concentration inequalities [61], we can establish high-probability bounds on the error $|P_i(D_t) - P(D_t)|$ between our estimate $P_i(D_t)$ and the true expected value $P(D_t)$. To illustrate how we estimate δ, let's consider the demographic parity ratio as defined in Equation 4.5. Here, p_A and p_B represent the sample probabilities of a favorable outcome (e.g., being hired) for groups A and B generated by pipeline P. Similarly, p'_A and p'_B denote the probabilities of a favorable outcome for groups A and B generated by pipeline P_i. According to Equation 4.4, we have:

$$Pr[|(p_A - p_B) - (p'_A - p'_B)| \geq \epsilon] \leq \delta \tag{4.7}$$

Break down Equation 4.7 into two deviation events to apply McDiarmid's inequality [61]:

$$Pr[|p_A - p'_A| \geq \epsilon_A] \leq exp(\frac{-2\epsilon_A^2}{\sum_{j=1}^{n} c_{ij}^2}) \quad and \quad Pr[|p_B - p'_B| \geq \epsilon_B] \leq exp(\frac{-2\epsilon_B^2}{\sum_{j=1}^{n} c_{ij}^2}) \tag{4.8}$$

Where value c_{ij} is a bound of the sensitivity of f to changes in the j^{th} variable. Let $x_1, \ldots, x_n$ be independent random variables taking values in some space. Consider a function $f : x^n \to \mathbb{R}$ satisfies the bounded differences property: for each $i \in 1, \ldots, n$. The value c_{ij} is estimated as follows [61]:

$$\sup_{x_1,\ldots,x_j,x'_j} |f(x_1, \ldots, x_j, \ldots, x_n) - f(x_1, \ldots, x'_j, \ldots, x_n)| \leq c_{ij} \tag{4.9}$$

To estimate δ, we bound the probability of the ratio deviating [11, 14]. To detect the violation of constraint, We consider the worst-case scenario where either p'_A or p'_B (or both) deviate from their expected values. The union bound allows us to combine the probabilities of these individual events to provide an upper bound on the total probability:

$$Pr[|(p_A - p_B) - (p'_A - p'_B)| \geq \epsilon] \leq Pr[|p_A - p'_A| \geq \epsilon_A] + Pr[|p_B - p'_B| \geq \epsilon_B] \tag{4.10}$$

To solve Equation 4.10, we set $\epsilon_A = \epsilon_B = \epsilon/2$, leading to $\delta = 2\exp\left(\frac{-\epsilon^2}{2\sum_{j=1}^{n} c_{ij}^2}\right)$. The following equations describe this relationship:

$$\begin{aligned} Pr[|p_A - p'_A| \geq \epsilon_A] + Pr[|p_B - p'_B| \geq \epsilon_B] &\leq 2exp(\frac{-\epsilon^2}{2\sum_{j=1}^{n} c_j^2}) \\ \Leftrightarrow Pr[|P_i(D_t) - P(D_t)| \geq \epsilon] &\leq 2exp(\frac{-\epsilon^2}{2\sum_{j=1}^{n} c_{ij}^2}) \end{aligned} \tag{4.11}$$

The sensitivity measure c_{ij} quantifies the maximum change in the output due to a change in the j^{th} input variable. It indicates the extent to how fairness value can change when only one input variable is modified while all others remain fixed. Therefore, the challenge lies in determining c_{ij} in a way that makes it represent the characteristic of preprocessor p_i and the compatibility of p_i with other processors. To achieve this goal, let us define an original training dataset $D_t = (x, y)$, which is passed through the first preprocessor p_1, resulting in the transformed dataset $D_{t1} = (x_1, y_1)$. Similarly, after the preprocessor p_i processes the data, we obtain the transformed dataset $D_{ti} = (x_i, y_i)$. To check the effectiveness of each preprocessor p_i, we evaluate and analyze its performance on the input dataset D_{ti}. Since p_i is the final step in generating D_{ti} before fitting the data D_{ti} to the base model m, we have the following equation:

$$P_i(D_t) = m(D_{ti}) \tag{4.12}$$

The dataset D_{ti} contains n non-protected attributes, denoted as $e_{i1}, \ldots, e_{in}$. Let's consider these data attributes as random variables for the concentration inequality. We can estimate the value of c_{ij} by assessing the impact of a preprocessor on the attribute e_{ij}. We treat data features as random variables to estimate sensitivity values, recognizing that an ML preprocessor inevitably modifies these features. These modifications can either reduce or amplify bias. Thus, by analyzing the changes in data features, we can effectively capture the characteristics and impact of the preprocessor. The sensitivity value c_{ij} is achieved by modifying e_j to make it fairer with respect to the protected attribute. The sensitivity value is then estimated as follows:

$$c_{ij} = m(e_{i1}, \ldots, e_{ij}, \ldots, e_{in}) - m(e_{i1}, \ldots, e'_{ij}, \ldots, e_{in}) \tag{4.13}$$

Now, we use c_{ij} as a sensitivity measure for concentration inequalities. This sensitivity measure c_{ij} represents the characteristics of the preprocessor p_i. Since applying p_i modifies the properties of data $D_{t,i-1}$ to generate new data D_{ti}, the data D_{ti} will inherently reflect the characteristics of p_i. By systematically modifying each feature in D_{ti} and breaking the relationship between the feature and the target, we can determine the extent to which the model relies on that particular feature. These feature scores thus capture the characteristics of p_i. It is important to note that this algorithm does not specify the impact of p_i on the original dataset directly. Instead, it specifies the impact of p_i on the dataset that has already been processed by the previous $i-1$ preprocessors. By doing so, we can assess how well p_i is compatible with the preprocessors that have already been applied. Furthermore, by focusing only on the preprocessors from p_1 to p_i and disregarding those after p_i, we can specify the modular fairness of p_i without disrupting the entire pipeline.

However, there are two main challenges that need to be addressed to apply concentration inequalities for checking modular fairness. First, we need to determine how to estimate c_{ij}. Second, concentration inequalities use the absolute value of the difference between the estimated value and the true expected value, $|P_i(D_t) - P(D_t)|$. If we cannot ascertain whether $P_i(D_t) > P(D_t)$ or $P_i(D_t) < P(D_t)$, the checking algorithm might produce a false positive when $P_i(D_t) > P(D_t)$. The following section will explain how we address these two problems to check modular fairness.

4.6.3 Fairness-Aware Feature Importance for Modules

This section describes how we estimate the contribution of each feature in a dataset to a fairness problem by mutating each feature to find its importance score, also known as the sensitivity c_{ij}.

Algorithm 6 Mutation Feature Importance

```
Require: A dataset $D_{ti} = (x, z, y)$, where $x$ represents the non-protected attributes, $z$ denotes the protected attribute, and $y$ is the label. A base model $m$.
1:  $b_i = m(D_{ti})$                          ▷ Compute the fairness score $b$
2:  important_scores = []
3:  $\Delta_i = 0$
4:  for each $j \in x$ do                      ▷ $j$ is a non-protected attribute
5:      $D'_{ti} = \text{mutate}(D, j)$  ▷ Mutate column j of $D_{ti}$ (Algorithm 7) to create a fairer version of $D$
6:      $b'_i = m(D'_{ti})$                    ▷ Compute the new fairness score $b_j$
7:      $c_{ij} = b'_i - b_i$                  ▷ Compute the feature score of feature $j$
8:      $\Delta_i$ += $c_{ij}$
9:      important_scores.add($c_{ij}$)
10: return important_scores, $\Delta_i$
```

4.6.3.1 Bias Detection Permutation Analysis

The Algorithm 6 is used to estimate c_{ij}. In line 1, we estimate the fairness score using the original data and the base model. Users can modify the metric used to calculate the score, allowing our algorithm to support various types of fairness metrics. Given a dataset D_{ti}, we capture the bias value b of this data using a base model m (line 2). After obtaining the fairness value b_i for the dataset D_{ti}, each feature of the data is modified to create a fairer version, denoted as D'_{ti} (lines 5-6). This modified dataset D'_{ti} is then fitted into the base model m to obtain a new fairness value b'_i. The feature score c_{ij} is estimated by calculating the difference between b_i and b'_i. If $c_{ij} < 0$, it indicates that feature j has been negatively impacted by the preprocessor p_i. Conversely, if $c_{ij} > 0$, it suggests that feature j has been positively impacted by p_i. Thus, the feature score c_{ij} estimated by our approach not only detects modular fairness but also helps to explain underlying bias issues introduced by the preprocessor.

The estimation Δ_i from Algorithm 6 is the sum of all feature scores c_{ij} for the dataset D_{ti}. Since c_{ij} is calculated using fairness metrics, it represents how effectively each feature contributes to the fairness of the final classification. Suppose we have obtained Δ_{i-1} at preprocessor P_{i-1} and now obtain Δ_i at preprocessor P_i. If $\Delta_i > \Delta_{i-1}$, it indicates that preprocessor P_i has reduced bias compared to P_{i-1}. This approach allows us to determine whether preprocessor p_i has increased or decreased bias. Thus, we can assess whether $P_i(D_t) > P(D_t)$ or $P_i(D_t) < P(D_t)$ by comparing Δ_i

and Δ_{i-1}. Note that a smaller value indicates a better outcome:

$$\begin{cases} P(D_t) \geq P_i(D_t), & \Delta_i \geq \Delta_{i-1} \\ P(D_t) < P_i(D_t), & \Delta_i < \Delta_{i-1} \end{cases} \tag{4.14}$$

If a user wants to check whether a preprocessor p_i can increase fairness, as in Example 4.1, our algorithm will determine whether $P_i(D_t) > P(D_t)$ or $P_i(D_t) < P(D_t)$. If $P(D_t) < P_i(D_t)$, we will skip the concentration inequality process for checking fairness and instead estimate the feature score, providing the user with a violation message. If $P(D_t) \geq P_i(D_t)$, we will use concentration inequalities to check modular fairness, checking if it can achieve high probability bounds. If the probability is not sufficiently high, we will issue a violation message to the user.

Algorithm 7 Bias Remover Mutation

```
Require: A dataset $D_{ti} = (x, z, y)$, where $x$ represents the non-protected attributes, $z$ denotes the
    protected attribute, $y$ is the labe. and a targeted non-protected attribute $x_i \in x$.
1: $x_i = \frac{x_i - \mu_i}{\sigma_i}$                                   ▷ Normalize feature to zero mean, unit variance
2: $\mu_0, \mu_1 = \frac{1}{|X_0|} \sum_{x \in X_0} x, \frac{1}{|X_1|} \sum_{x \in X_1} x$      ▷ Centroid of group with $z = 0$ and $z = 1$
3: $\delta = \mu_1 - \mu_0$                                   ▷ Vector difference between centroids
4: $\tilde{x}_i = x_i - \lambda \cdot (S_i \cdot \delta)$                   ▷ Adjust based on sensitive attribute
5: Return $\tilde{x_i}$                                         ▷ Transformed dataset
```

4.6.3.2 Bias Remover Mutation

The bias remover mutation algorithm is designed to adjust feature values in the dataset so that the outcomes are not influenced by sensitive attributes. A detailed description of this method has been shown in the Algorithm 7. The first step normalizes each feature to have zero mean and unit variance (Line 1). This step is essential to ensure that the features are on the same scale, which helps in fair adjustments [33]. The centroids of the data points belonging to each group defined by the sensitive attribute (e.g., $S = 0$ and $S = 1$) are calculated (Line 2). These centroids represent the "center" of the data for each group. The vector difference between the centroids of the two groups is computed. This vector indicates how much one group differs from the other (Line 3). The targeted feature is adjusted by shifting the values along the difference vector scaled by the parameter λ (Line 4). This step effectively reduces the impact of the sensitive attribute on the outcome. The transformed feature $\tilde{X}$ is returned, which ideally should improve fairness compared to the original one (Line 5) [33].

4.7 Evaluation

In this section, we describe the design of the experiments to evaluate the efficiency of *Fairness Checker*. Since no existing method can simultaneously detect and explain the unfairness of pipeline modules, we compare our approach with bias mitigation algorithms that either detect or mitigate bias. Additionally, we evaluate the specific contributions of the proposed method, including the application of concentration inequalities to detect bias and fairness-aware feature importance.

RQ1: How effective is *Fairness Checker* for detecting modular fairness bugs? To address this question, we quantify the number of fairness bugs that *Fairness Checker* is able to detect compared to bias detection methods, allowing us to assess the capability of concentration inequality to detect modular fairness bugs. We also compare the average runtime required to detect a bug using *Fairness Checker* with that of existing methods.

RQ2: How efficient are concentration inequalities for checking modular fairness specification? We answer this question by applying the difference in fairness score of the ML pipeline with and without a specified ML component to the concentration inequalities.

RQ3: Are the fairness-aware feature importance technique effective in estimating the fairness impact of the features? To address this question, we compare the pipeline's fairness score after excluding negative (unfair) features identified by our method with traditional feature permutation techniques. Additionally, we compare the average runtime for estimating feature scores using *Fairness Checker* against that of existing methods.

RQ4: How does fairness checker generalize for various datasets, metrics, and ML algorithms? We address this question by evaluating the efficiency of the fairness checker across a wide range of datasets, different types of fairness metrics, and various machine learning algorithms.

4.7.1 Experimental Setup

4.7.1.1 Benchmark

Dataset We use four popular datasets for our evaluation which are used in prior works [20, 68, 69]:

The **Adult Census** (race) [49] contains 32,561 records and 12 features from the 1994 U.S. census, aiming to predict whether an individual earns over 50K annually.

The **Bank Marketing** (age) [50] includes 41,188 records with 20 features from direct marketing campaigns by a Portuguese bank, aiming to predict if a client will subscribe to a term deposit.

The **German Credit** (sex) [51] contains 1,000 records with 21 features to predict good/bad credit.

The **Titanic** (sex) [52] has 891 data points with ten features containing individual information about Titanic passengers. The dataset is used to predict who survived the Titanic shipwreck.

Fairness Bug We evaluated our method using real-world programs from a recent study [17]. Our benchmark consisted of 45 buggy pipelines representing diverse ML pipeline structures and preprocessing algorithms. Of these 45 cases, 24 involved bugs in data preprocessing components, 11 in feature engineering components, and 10 in classifier components. We evaluated the *Fairness Checker* method, which integrates various types of fairness annotations, the concentration inequality, and the feature mutation technique, on all 45 buggy cases. Concentration inequalities and fairness-aware feature importance techniques are currently applicable only to preprocessing algorithms, as demonstrated by the 24 buggy cases in data preprocessing and 11 buggy cases in feature engineering. Consequently, we restrict our evaluation of the effectiveness of these techniques to those specific buggy cases.

4.7.1.2 Methods Evaluated for Comparison

Bias mitigation methods (RQ1) To evaluate the general effectiveness of *Fairness Checker*, we compare it with various fairness bug-fixing methods [62], including: *Reweighing* (**R**) [53], *Disparate Impact Remover* (**DIR**) [33], *Parfait-ML* (**PML**) [67], *Equalized Odds* (**EO**) [42], *FaX-AI* (**FAX**) [41], *Reject Option Classification* (**ROC**) [54]. When an ML pipeline contains bugs, we assume these bugs affect all fairness metrics. Moreover, since these mitigation methods aim to address issues holistically, we assume that all fairness metrics will be improved once a bias issue is resolved using these techniques. We use two fairness metrics, SPD and EOD, so we consider a bias mitigation technique to successfully detect a bias bug if it improves both metrics for the buggy ML pipeline.

Fair preprocessing (RQ2) To assess the efficiency of concentration inequalities in specifying modular fairness, we leverage the evaluation methodology proposed in *Fair Preprocessing* [17]. Specifically, for a component where the developer aims to specify fairness, we run the ML pipeline with and without the component 10 times, recording the mean fairness score between the two versions of the pipeline. We then use this mean value as ϵ, setting the threshold δ at 0.85, meaning that violations should only be flagged when the confidence level is ≥ 0.85, which is commonly used in fairness research [11, 14].

Feature importance methods (RQ3) our fairness-aware feature importance (FA) is inspired by the feature permutation (FP) methods [35], which explores model-agnostic approaches for determining feature importance by using the permutation importance technique. FP is also implemented as the *permutation_importance* API in *Scikit-learn* [2]. Thus, we evaluate the effectiveness of our fairness-aware feature importance by comparing it with FP. Specifically, after obtaining the feature scores, we exclude the top three negatively ranked features identified by each method, rerun the ML pipeline, and then compare the resulting scores.

4.7.2 Effectiveness of Fairness Bug Detection (RQ1)

To evaluate the effectiveness of *Fairness Checker*, we measure the number of bugs it detects compared to other bias mitigation techniques. We assess *Fairness Checker* by benchmarking its performance against seven fairness bug detection methods across four fairness datasets, covering 45

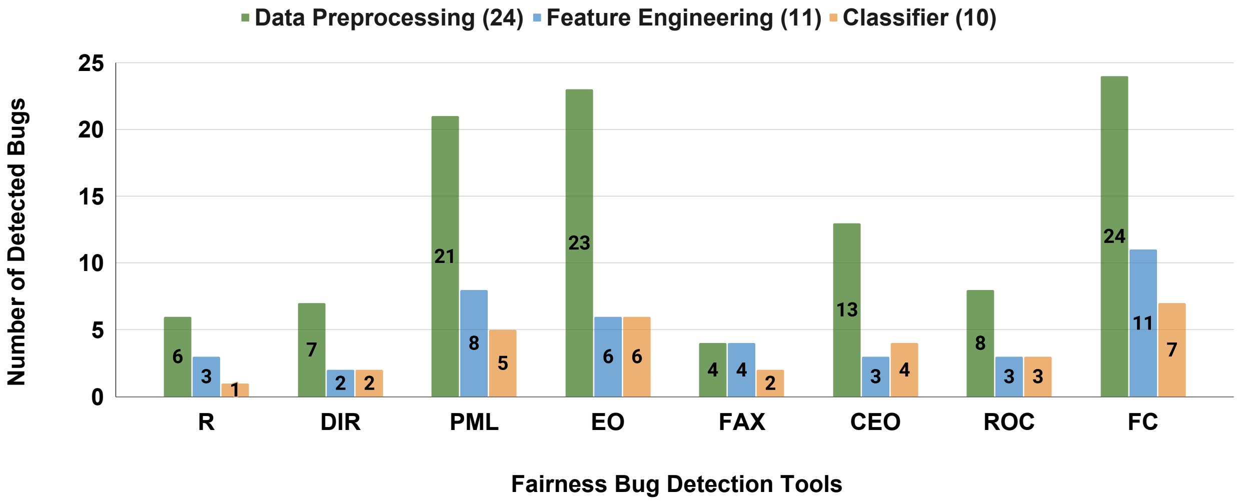


Figure 4.3: Comparison of detected bugs between *Fairness Checker* (FC) and fairness detection techniques.

buggy cases. Each case represents a faulty component in an ML pipeline, categorized into three areas: data preprocessing, feature engineering, and classifier. The benchmark has four types of data preprocessing algorithms, including encoding, sampling, scaling, and imputation. The benchmark includes three types of feature engineering: select K best, principal component analysis, and custom feature selection. Finally, the benchmark includes six types of classifiers: logistic regression, random forest, decision tree, support vector machine, k-neighbors, and XGBoost.

Figure 4.3 illustrates the number of detected fairness bugs across various ML pipeline components using different bug detection techniques. The figure illustrates that *Fairness Checker* identified 42 out of 45 fairness bugs (93%), whereas the most effective bias mitigation techniques detected up to 35 out of 45 bugs (78%) in total. This highlights *Fairness Checker*'s superior ability to detect fairness bugs. Furthermore, *Fairness Checker* successfully identified all 24 fairness bugs (100%) in data preprocessing algorithms and all 11 fairness bugs (100%) in feature engineering algorithms. *Fairness Checker* can detect more fairness bugs because it is comprehensive enough to identify fairness issues across different types of ML algorithms. Each time a developer checks a component of the ML pipeline, *Fairness Checker* focuses solely on that component to determine whether it violates fairness conditions. This modular approach is one of the key advantages of our method, allowing it to concentrate on specific components to identify fairness bugs rather than

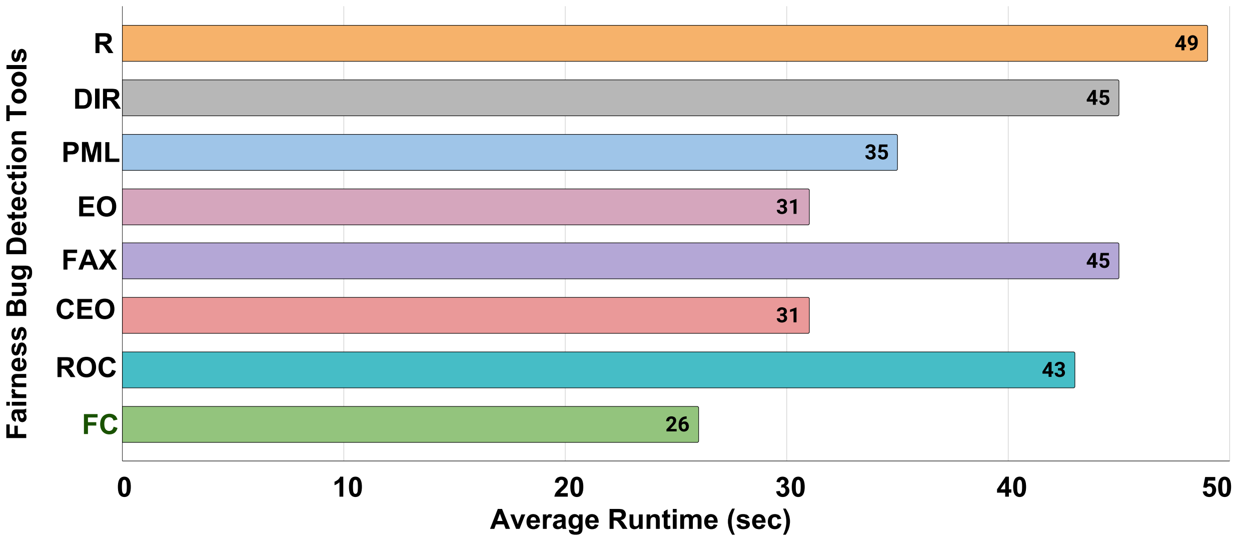


Figure 4.4: Average runtime of bias detection tools

analyzing the entire pipeline. In other words, our technique can effectively localize the root cause of the bias issue. In addition, the developer can apply our technique in the design time. In contrast, current fairness bug detection techniques target the entire pipeline and are applicable in post-development. Moreover, existing fairness bug-fixing techniques are limited in their scope, focusing on individual ML pipeline components like data preprocessing or modeling. This narrow focus restricts their ability to detect and address a wider range of fairness issues. For instance, the k-neighbors (KNN) algorithm can produce biased predictions if the data is not scaled before use. This is because KNN relies on distance calculations, and features with larger numerical ranges can unduly influence the outcome. While *Fairness Checker* can identify this problem by checking for varying numerical ranges in the input data, traditional bug-fixing techniques often overlook this issue, concentrating instead on tuning the KNN algorithm itself. This explains why some existing bias detection methods, such as R or FAX, report a low number of detected bugs in data preprocessing algorithms, as shown in Figure 4.3.

Figure 4.4 presents the average runtime to detect the fairness bug of each bias detection technique. On average, the runtime for detecting a bug with *Fairness Checker* is faster than traditional fairness bug detection methods. Specifically, *Fairness Checker* requires only around 26

Table 4.3: Effectiveness of concentration inequalities

Dataset	# Buggy Cases	# Correct Detection	Detection Accuracy
Adult Census	9	8	88.87%
Bank Marketing	12	10	83.34%
German Credit	6	6	100.00%
Titanic	8	7	87.50%

seconds to detect a fairness bug, compared to 31 seconds for the most efficient bias detection technique. This increased efficiency is because *Fairness Checker* focuses on modular fairness to detect fairness bugs early in the buggy program and halts execution before completion. While additional time is required to ensure that the contract conditions are met, this process is still generally faster than executing the remaining steps of the machine learning (ML) pipeline, particularly the training phase. The training phase can be especially time-consuming, depending on the complexity of the model, the size of the dataset, and the computational resources available. Moreover, when bias mitigation techniques are applied, they often introduce even more overhead. Many of these techniques require either additional training, retraining of the model, or pre-processing of the data to transform it in ways that reduce bias.

4.7.3 Efficiency of the Concentration Inequalities (RQ2)

According to Equation 4.4, evaluating the effectiveness of concentration inequalities in detecting fairness bugs requires determining the value of ϵ. In this evaluation, we calculate ϵ for each buggy pipeline by measuring the difference in fairness scores between the pipeline with and without the buggy preprocessor. Specifically, for each of the 34 buggy cases involving the preprocessor, we execute both versions of the buggy pipeline 10 times and record the average difference. This average difference is used as ϵ to write fairness annotators for evaluating concentration inequalities. Developers need input ϵ to specify fairness; thus, we calculate ϵ to define the specifications in the first step of the evaluation.

Following the first step, we assess the effectiveness of our approach using the calculated ϵ values from the previous step. Table 4.3 shows the detection accuracy of concentration inequalities in identifying modular fairness. There are 35 buggy cases in total, with 9 from the Adult Census dataset, 12 from the Bank Marketing dataset, 6 from the German Credit dataset, and 7 from the Titanic dataset. Notably, *Fairness Checker* achieves 88.87%, 84.34%, 100%, and 87.5% detection accuracy on the Adult Census, Bank Marketing, German Credit, and Titanic datasets, respectively. Our method demonstrates high detection accuracy because we leverage feature scores as sensitivity values for concentration inequalities. This idea is inspired by *Fair Processing*, which has shown that a preprocessor can introduce bias into an ML pipeline, as it may unintentionally modify features in a way that disadvantages protected attributes. For example, a *StandardScaler* standardizes features by removing the mean and scaling them to unit variance. Still, it is sensitive to extreme values because it relies on both the mean and the standard deviation, which are heavily influenced by outliers. In datasets like the Adult Census, where the *salary* feature contains many extreme values that are crucial for fair predictions, standard scaling can inadvertently diminish the significance of these values. This reduction in significance can introduce a new bias into the model.

4.7.4 Fairness-aware Feature Importance (RQ3)

To assess the performance of our fairness-aware feature importance technique compared to the existing feature permutation technique (*FP*) [35], We use SPD and EOD, both widely adopted in fairness research [48, 14], as the scoring functions for both methods. First, we apply each technique to obtain feature scores. Negative scores indicate that the corresponding features introduce bias into the ML pipeline, while positive scores suggest they help reduce bias. To evaluate the effectiveness of the feature importance methods, we remove the features with negative scores from the dataset. We then rerun the pipeline on this modified dataset. If the bias is reduced, it confirms that the feature importance technique has accurately estimated the negative and positive feature contributions.

The Tables 4.4, 4.5, 4.6, 4.7 present the bias scores of buggy ML pipelines on the original dataset (Org), on a modified dataset based on our fairness-aware feature importance technique (FA), and on a modified dataset based on the traditional feature permutation technique (FP). Lower bias scores indicate better performance. As shown in these tables, FA achieves the best bias

Table 4.4: Effectiveness of fairness-aware feature importance on Adult Census (AC)

Tool	Metric	AC1	AC2	AC3	AC4	AC5	AC6	AC7	AC8	AC9	AVG
Org	**SPD**	**0.024**	0.102	0.103	**0.103**	0.087	0.842	0.124	0.104	0.065	*0.173*
FA		0.096	**0.100**	0.100	0.104	**0.087**	**0.765**	**0.056**	**0.051**	**0.061**	*0.158*
FP		0.101	0.126	**0.095**	0.113	0.134	0.802	0.073	0.140	0.072	*0.184*
Org	**EOD**	0.123	0.072	0.031	**0.034**	0.045	0.144	0.063	0.023	**0.047**	*0.065*
FA		**0.059**	**0.065**	**0.014**	0.041	**0.045**	**0.075**	**0.060**	**0.011**	0.057	*0.047*
FP		0.095	0.097	0.034	0.046	0.067	0.129	0.064	0.030	0.048	*0.068*

Org: original dataset, FC: dataset with features mutated by FA, FD: dataset with features mutated by FD, AVG: average feature score for each tool. **Bold** values indicate the best fairness score, and *italic* values represent averages.

Table 4.5: Effectiveness of fairness-aware feature importance on Bank Marketing (BM)

Tool	Metric	BM1	BM2	BM3	BM4	BM5	BM6	BM7	BM8	BM9	BM10	BM11	BM12	AVG
Org	**SPD**	0.113	0.096	0.094	0.098	0.101	0.095	0.076	0.116	0.118	**0.123**	0.054	0.052	*0.095*
FA		**0.093**	**0.081**	**0.063**	**0.069**	0.080	**0.041**	**0.057**	**0.065**	**0.117**	0.124	**0.055**	**0.051**	*0.075*
FP		0.106	0.104	0.081	0.098	**0.040**	0.093	0.093	0.136	0.128	0.134	0.058	0.052	*0.094*
Org	**EOD**	0.089	0.109	0.058	0.134	0.140	0.067	0.085	0.126	0.147	0.161	0.029	0.022	*0.097*
FA		**0.059**	**0.004**	**0.029**	**0.048**	0.103	**0.006**	**0.059**	**0.102**	**0.107**	0.144	0.027	**0.008**	*0.058*
FP		0.116	0.117	0.100	0.134	**0.007**	0.063	0.097	0.170	0.167	**0.124**	**0.007**	0.020	*0.094*

Org: original dataset, FC: dataset with features mutated by FA, FD: dataset with features mutated by FD, AVG: average feature score for each tool. **Bold** values indicate the best fairness score, and *italic* values represent averages.

scores in 26 out of 35 buggy cases for SPD, and 25 out of 35 cases for EOD. In contrast, the FP performs best in only 4 out of 35 cases for SPD, and 6 out of 35 cases for EOD. These results highlight that our approach is more effective than the traditional feature permutation method. Instead of randomly shuffling features like the traditional method, we deliberately mutate them to enhance fairness with respect to the protected attribute. By intentionally adjusting the features, we ensure that the new features are fairer and enable us to estimate feature scores more accurately. There are 5 out of 35 cases for SPD and 4 out of 35 cases for EOD, in which the original dataset achieved the best bias score. This occurs because, while individual features may introduce bias, combining them can reduce overall bias. Consequently, removing certain features might unintentionally disrupt this balance, leading to increased bias.

Figure 4.5 shows the average runtime of FA and FP on four datasets, including Adult Census, Bank Marketing, German Credit, and Titanic datasets. The figure indicates that our technique requires less average runtime to estimate feature scores than the traditional feature permutation method. For example, in the Adult Census dataset, the FA method takes only 34 seconds on

Table 4.6: Effectiveness of fairness-aware feature importance on German Credit (GC)

Tool	**Metric**	**GC1**	**GC2**	**GC3**	**GC4**	**GC5**	**GC6**	**Avg**
Org	**SPD**	0.025	0.007	0.020	**0.133**	0.067	0.159	*0.069*
FA		**0.008**	**0.004**	**0.000**	0.146	**0.062**	**0.122**	*0.057*
FP		0.017	0.015	0.004	0.213	0.062	0.146	*0.076*
Org	**EOD**	0.038	0.055	0.015	**0.028**	0.062	0.128	*0.054*
FA		**0.005**	0.048	**0.013**	0.032	0.056	**0.105**	*0.044*
FP		0.068	**0.041**	0.013	0.119	**0.051**	0.157	*0.075*

Org: original dataset, FC: dataset with features mutated by FA, FD: dataset with features mutated by FD, AVG: average feature score for each tool. **Bold** values indicate the best fairness score, and *italic* values represent averages.

Table 4.7: Effectiveness of fairness-aware feature importance on Titanic (TT)

Tool	**Metric**	**TT1**	**TT2**	**TT3**	**TT4**	**TT5**	**TT6**	**TT7**	**TT8**	**Avg**
Org	**SPD**	0.733	0.666	0.718	0.724	0.628	**0.599**	0.868	0.667	*0.700*
FA		**0.696**	**0.520**	**0.604**	0.686	**0.608**	0.686	**0.594**	0.662	*0.632*
FP		0.839	0.707	0.606	**0.646**	0.735	0.846	0.666	**0.620**	*0.708*
Org	**EOD**	0.667	0.589	0.653	0.686	0.596	**0.569**	0.851	0.654	*0.658*
FA		**0.556**	**0.321**	**0.514**	**0.645**	0.562	0.746	**0.519**	**0.504**	*0.546*
FP		0.857	0.535	0.592	0.667	**0.554**	0.837	0.572	0.572	*0.648*

Org: original dataset, FC: dataset with features mutated by FA, FD: dataset with features mutated by FD, AVG: average feature score for each tool. **Bold** values indicate the best fairness score, and *italic* values represent averages.

average to calculate the feature scores, whereas the FD method requires up to 145 seconds on average. The randomness of the original method requires multiple shuffles and averaging, which adds substantial computational overhead. In contrast, our method directly permutes features in a way that aligns them with the protected attribute, resulting in consistent feature scores while significantly reducing computational costs by only permuting the feature once.

4.7.5 Generalization (RQ4)

To assess *Fairness Checker*'s generalizability in detecting modular fairness, we tested it on four datasets, two fairness metrics, and 13 ML algorithms. The generalization capabilities of *Fairness Checker* are presented in Figure 4.3 and Tables 4.4, 4.5, 4.6, 4.7. These results demonstrate that *Fairness Checker* performs well across diverse datasets, fairness metrics, and ML algorithms. Tables 4.4, 4.5, 4.6, 4.7 show that *Fairness Checker* effectively localizes bias through feature scoring across diverse datasets and fairness metrics.

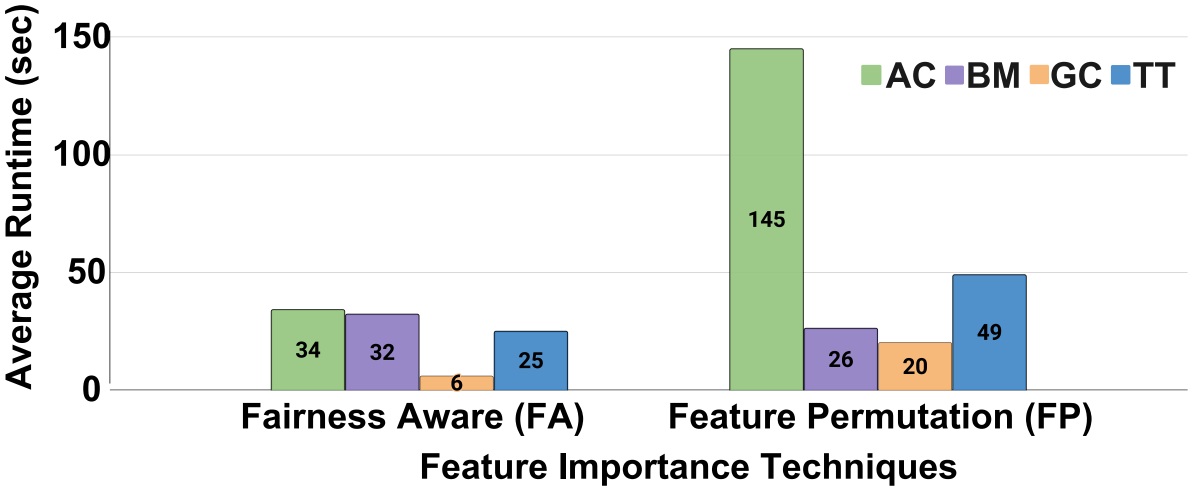


Figure 4.5: Average runtime of FA and FP

Fairness Checker demonstrates strong generalization across a wide range of fairness problems due to its flexible and data-agnostic approach. Different from traditional methods that often rely on specific characteristics of datasets, predefined fairness metrics, or the structure of particular machine learning (ML) algorithms [53, 33], *Fairness Checker* operates independently of these constraints. For example, Figure 4.3 shows that EO has up to 23 out of 24 detected bugs for data preprocessing algorithms; however, this method is only able to detect 6 out of 11 bugs for feature engineering algorithms and 6 out of 10 bugs for classifiers. Unlike existing methods, *Fairness Checker* achieves a high detection rate across various algorithms by analyzing both the input and output of ML models without relying on predefined fairness assumptions. By assessing feature importance, *Fairness Checker* identifies potential sources of bias in the data. It then applies concentration inequalities to determine if fairness deviations are statistically significant, enabling the detection of subtle, complex forms of bias often missed by conventional methods.

4.8 Threats to Validity

Fairness Checker has been evaluated primarily on tabular datasets and binary classification tasks using classical models like logistic regression, decision trees, and support vector machines. This focus has allowed for a thorough analysis of its performance in familiar contexts, demonstrating its effectiveness in detecting fairness bugs [18, 36, 9, 37, 69]. However, its applicability to other data types—such as images, text, or time series—and more complex models like deep neural networks and ensemble methods remain unexplored. Further research is needed to assess *Fairness Checker*'s adaptability to unstructured data. Despite this limitation, we believe that the concept of using contracts to detect fairness bugs is not restricted to tabular datasets and classical machine learning models. The core idea of formalizing fairness expectations through contracts and systematically checking models against these specifications can be extended.

Fairness Checker employs base models to estimate feature importance scores for fairness bug detection. This approach provides a computationally efficient way to approximate how individual features influence the model's predictions of unfairness. However, the base model may not capture all the complexities of the real, deployed model. This discrepancy could lead to less accurate estimations of feature importance, potentially affecting the precision of fairness bug detection. To address these concerns, we have conducted an extensive evaluation to assess the effectiveness of *Fairness Checker* across a wide range of datasets, fairness metrics, and machine learning algorithms. This comprehensive analysis demonstrates the robustness and versatility of *Fairness Checker* in detecting fairness issues under various conditions.

4.9 Conclusion

This work focuses on detecting and explaining *modular fairness* bugs modularly in ML pipelines, which are often overlooked by *global fairness* approaches. Existing methods struggle to detect fairness issues due to biases introduced by individual components in complex ML pipelines. Our approach introduces fairness annotators, allowing users to specify and check fairness constraints for preprocessing steps using concentration inequalities for strong guarantees. Experiments show that *Fairness Checker* outperforms other fairness bug detection methods. Future work will extend our approach to address fairness bugs in deep learning models, broadening the scope of this study.

Bibliography

[1] 2019. Decision Tree Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[2] 2019. Feature Permutation. https://scikit-learn.org/stable/modules/generated/sklearn.inspection.permutation_importance.html#sklearn.inspection.permutation_importance

[3] 2019. Gradient Boosting Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

[4] 2019. K-Neighbors Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[5] 2019. Linear SVC. https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[6] 2019. Logistic Regression. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[7] 2019. Random Forest Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[8] 2024. Fairness Checker. https://github.com/tess022095/Fairness-Checker

[9] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 625–635.

[10] Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. 2023. Design by Contract for Deep Learning APIs. In *ESEC/FSE'2023: The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, California).

[11] Aws Albarghouthi and Samuel Vinitsky. 2019. Fairness-Aware Programming. In *Proceedings of the Conference on Fairness, Accountability, and Transparency* (Atlanta, GA, USA) *(FAT* '19)*. Association for Computing Machinery, New York, NY, USA, 211–219. https://doi.org/10.1145/3287560.3287588

[12] Edesio Alcobaça, Felipe Siqueira, Adriano Rivolli, Luís P. F. Garcia, Jefferson T. Oliva, and André C. P. L. F. de Carvalho. 2020. MFE: Towards reproducible meta-feature extraction. *Journal of Machine Learning Research* 21, 111 (2020), 1–5. http://jmlr.org/papers/v21/19-348.html

[13] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2018. Themis: automatically testing software for discrimination. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 871–875. https://doi.org/10.1145/3236024.3264590

[14] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Probabilistic verification of fairness properties via concentration. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 118 (oct 2019), 27 pages. https://doi.org/10.1145/3360544

[15] Rachel KE Bellamy, Kuntal Dey, Michael Hind, Samuel C Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, et al. 2018. AI Fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *arXiv preprint arXiv:1810.01943* (2018).

[16] Reuben Binns. 2018. Fairness in machine learning: Lessons from political philosophy. In *Conference on Fairness, Accountability and Transparency*. PMLR, 149–159.

[17] Sumon Biswas and Hridesh Rajan. 2021. Fair preprocessing: towards understanding compositional fairness of data transformers in machine learning pipeline. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 981–993. https://doi.org/10.1145/3468264.3468536

[18] Sumon Biswas and Hridesh Rajan. 2023. Fairify: Fairness Verification of Neural Networks. In *ICSE'2023: The 45th International Conference on Software Engineering* (Melbourne, Australia).

[19] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2091–2103. https://doi.org/10.1145/3510003.3510057

[20] Miranda Bogen and Aaron Rieke. 2018. Help wanted: an examination of hiring algorithms, equity, and bias. https://api.semanticscholar.org/CorpusID:158203520

[21] Ajay Byanjankar, Markku Heikkilä, and Jozsef Mezei. 2015. Predicting credit risk in peer-to-peer lending: A neural network approach. In *2015 IEEE symposium series on computational intelligence*. IEEE, 719–725.

[22] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data mining and knowledge discovery* 21, 2 (2010), 277–292.

[23] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in machine learning software: Why? how? what to do?. In *Proceedings of the 29th ACM Joint Meeting on*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 429–440.

[24] Joymallya Chakraborty, Suvodeep Majumder, Zhe Yu, and Tim Menzies. 2020. Fairway: a way to build fair ML software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 654–665.

[25] Zhenpeng Chen, Jie M. Zhang, Max Hort, Mark Harman, and Federica Sarro. 2024. Fairness Testing: A Comprehensive Survey and Analysis of Trends. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 137 (jun 2024), 59 pages. https://doi.org/10.1145/3652155

[26] Zhenpeng Chen, Jie M Zhang, Federica Sarro, and Mark Harman. 2022. A comprehensive empirical study of bias mitigation methods for software fairness. *arXiv preprint arXiv:2207.03277* (2022).

[27] Zhenpeng Chen, Jie M Zhang, Federica Sarro, and Mark Harman. 2022. MAAT: a novel ensemble approach to addressing fairness and performance bugs for machine learning software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1122–1134.

[28] Alexandra Chouldechova. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data* 5, 2 (2017), 153–163.

[29] Alexandra Chouldechova, Diana Benavides-Prado, Oleksandr Fialko, and Rhema Vaithianathan. 2018. A case study of algorithm-assisted decision making in child maltreatment hotline screening decisions. In *Conference on Fairness, Accountability and Transparency*. PMLR, 134–148.

[30] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O'Donoghue, Daniel Visentin, et al. 2018. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine* 24, 9 (2018), 1342–1350.

[31] Virginia Eubanks. 2018. *Automating inequality: How high-tech tools profile, police, and punish the poor*. St. Martin's Press.

[32] Michael Feffer, Martin Hirzel, Samuel C Hoffman, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2022. An Empirical Study of Modular Bias Mitigators and Ensembles. *arXiv preprint arXiv:2202.00751* (2022).

[33] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 259–268.

[34] Anthony Finkelstein, Mark Harman, S. Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. 2008. "Fairness Analysis" in Requirements Assignments. In *2008 16th IEEE International Requirements Engineering Conference*. 115–124. https://doi.org/10.1109/RE.2008.61

[35] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. 2019. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *Journal of Machine Learning Research* 20, 177 (2019), 1–81.

[36] Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. 2019. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the conference on fairness, accountability, and transparency*. 329–338.

[37] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. 498–510.

[38] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software*

Engineering (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 498–510. https://doi.org/10.1145/3106237.3106277

[39] Xuanqi Gao, Juan Zhai, Shiqing Ma, Chao Shen, Yufei Chen, and Qian Wang. 2022. FairNeuron: improving deep neural network fairness with adversary games on selective neurons. In *Proceedings of the 44th International Conference on Software Engineering*. 921–933.

[40] Usman Gohar, Sumon Biswas, and Hridesh Rajan. 2023. Towards Understanding Fairness and its Composition in Ensemble Machine Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1533–1545. https://doi.org/10.1109/ICSE48619.2023.00133

[41] Przemyslaw A Grabowicz, Nicholas Perello, and Aarshee Mishra. 2022. Marrying fairness and explainability in supervised learning. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. 1905–1916.

[42] Moritz Hardt, Eric Price, and Nati Srebro. 2016. Equality of opportunity in supervised learning. *Advances in neural information processing systems* 29 (2016).

[43] Moritz Hardt, Eric Price, and Nathan Srebro. 2016. Equality of opportunity in supervised learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3323–3331.

[44] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the eighth international workshop on data mining for online advertising*. 1–9.

[45] Mitchell Hoffman, Lisa B Kahn, and Danielle Li. 2018. Discretion in hiring. *The Quarterly Journal of Economics* 133, 2 (2018), 765–800.

[46] Max Hort, Zhenpeng Chen, Jie M. Zhang, Mark Harman, and Federica Sarro. 2024. Bias Mitigation for Machine Learning Classifiers: A Comprehensive Survey. *ACM J. Responsib. Comput.* 1, 2, Article 11 (jun 2024), 52 pages. https://doi.org/10.1145/3631326

[47] Max Hort and Federica Sarro. 2021. Did you do your homework? Raising awareness on software fairness and discrimination. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1322–1326.

[48] Max Hort, Jie M Zhang, Federica Sarro, and Mark Harman. 2021. Fairea: A model behaviour mutation approach to benchmarking bias mitigation methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 994–1006.

[49] Kaggle. 2017. Adult Census Dataset.
https://www.kaggle.com/datasets/uciml/adult-census-income

[50] Kaggle. 2017. Bank Marketing Dataset. https://www.kaggle.com/c/bank-marketing-uci

[51] Kaggle. 2017. German Credit Dataset.
https://www.kaggle.com/datasets/uciml/german-credit

[52] Kaggle. 2017. Titanic ML Dataset. https://www.kaggle.com/c/titanic/data

[53] Faisal Kamiran and Toon Calders. 2012. Data preprocessing techniques for classification without discrimination. *Knowledge and information systems* 33, 1 (2012), 1–33.

[54] Faisal Kamiran, Asim Karim, and Xiangliang Zhang. 2012. Decision theory for discrimination-aware classification. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 924–929.

[55] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. 2012. Fairness-aware classifier with prejudice remover regularizer. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 35–50.

[56] Konstantina Kourou, Themis P Exarchos, Konstantinos P Exarchos, Michalis V Karamouzis, and Dimitrios I Fotiadis. 2015. Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal* 13 (2015), 8–17.

[57] Oliver Kramer and Oliver Kramer. 2016. Scikit-learn. https://scikit-learn.org/

[58] Yanhui Li, Linghan Meng, Lin Chen, Li Yu, Di Wu, Yuming Zhou, and Baowen Xu. 2022. Training data debugging for the fairness of machine learning software. In *Proceedings of the 44th International Conference on Software Engineering*. 2215–2227.

[59] Milad Malekipirbazari and Vural Aksakalli. 2015. Risk assessment in social lending via random forests. *Expert Systems with Applications* 42, 10 (2015), 4621–4631.

[60] Pranav Maneriker, Codi Burley, and Srinivasan Parthasarathy. 2023. Online Fairness Auditing through Iterative Refinement. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 1665–1676. https://doi.org/10.1145/3580305.3599454

[61] Colin McDiarmid et al. 1989. On the method of bounded differences. *Surveys in combinatorics* 141, 1 (1989), 148–188.

[62] Giang Nguyen, Sumon Biswas, and Hridesh Rajan. 2023. Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/3611643.3616257

[63] Luca Oneto, Michele Donini, Andreas Maurer, and Massimiliano Pontil. 2019. Learning fair and transferable representations. *arXiv preprint arXiv:1906.10673* (2019).

[64] Dino Pedreshi, Salvatore Ruggieri, and Franco Turini. 2008. Discrimination-aware data mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 560–568. https://doi.org/10.1145/1401890.1401959

[65] Claudia Perlich, Brian Dalessandro, Troy Raeder, Ori Stitelman, and Foster Provost. 2014. Machine learning for targeted display advertising: Transfer learning in action. *Machine learning* 95, 1 (2014), 103–127.

[66] Guanhong Tao, Weisong Sun, Tingxu Han, Chunrong Fang, and Xiangyu Zhang. 2022. RULER: discriminative and iterative adversarial training for deep neural network fairness. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1173–1184.

[67] Saeid Tizpaz-Niari, Ashish Kumar, Gang Tan, and Ashutosh Trivedi. 2022. Fairness-aware Configuration of Machine Learning Libraries. *arXiv preprint arXiv:2202.06196* (2022).

[68] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 401–416.

[69] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 98–108.

[70] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 98–108. https://doi.org/10.1145/3238147.3238165

[71] Rhema Vaithianathan, Tim Maloney, Emily Putnam-Hornstein, and Nan Jiang. 2013. Children in the public benefit system at risk of maltreatment: Identification via predictive modeling. *American journal of preventive medicine* 45, 3 (2013), 354–359.

[72] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rogriguez, and Krishna P Gummadi. 2017. Fairness constraints: Mechanisms for fair classification. In *Artificial Intelligence and Statistics*. PMLR, 962–970.

[73] Brian Hu Zhang, Blake Lemoine, and Margaret Mitchell. 2018. Mitigating unwanted biases with adversarial learning. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*. 335–340.

[74] Hantian Zhang, Xu Chu, Abolfazl Asudeh, and Shamkant B. Navathe. 2021. OmniFair: A Declarative System for Model-Agnostic Group Fairness in Machine Learning. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2076–2088. https://doi.org/10.1145/3448016.3452787

[75] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 949–960.

CHAPTER 5. GENERAL CONCLUSION

In this dissertation, we place a strong emphasis on optimizing fairness and detecting bias within the ML pipeline. While much of the existing research in this area has focused on mitigating and detecting bias, these efforts often overlook several critical aspects. Specifically, many approaches fail to address the fairness-accuracy tradeoff, which is a crucial consideration when balancing model performance with fairness objectives. Moreover, current methods fall short in terms of fairness bug localization—accurately identifying where bias occurs within the model or pipeline—and fairness bug explanation, which is essential for understanding the root causes of bias and developing effective solutions. By tackling these overlooked challenges, this work aims to provide a more holistic and robust approach to fairness in ML. First, we introduce *Fair-AutoML*, a cutting-edge system that improves current AutoML frameworks by addressing fairness issues while balancing the fairness-accuracy trade-off. The main idea behind *Fair-AutoML* is to adjust the hyperparameters of problematic models to fix fairness-related problems. Second, we introduce *Fairness Contract*, an innovative contract-based framework designed to detect and pinpoint fairness bugs throughout the ML pipeline. This approach is both flexible and adaptable, enabling the capture of fairness properties across various ML algorithms. Third, we propose *Fairness Checker* focusing on detecting and explaining modular fairness bugs within machine learning (ML) pipelines. While many current methods prioritize global fairness, they often fail to address modular fairness, making it difficult to trace and explain the origins of fairness issues. Our approach introduces fairness annotators, which allow users to specify whether a particular preprocessing step contributes to improving the fairness of the ML pipeline. These annotators use concentration inequalities to verify developer-specified conditions, offering robust soundness guarantees.

5.1 Future Work

5.1.1 Fairness Bugs Repairing for Deep Learning (DL) Systems

Along with bias mitigation techniques for classical ML algorithms and tabular datasets, we need tools and methods to repair unfairness defects in DL systems. Our works showed the prospect for fairness defect repair in ML-based software using AutoML. For example, *Fair-AutoML* [1] showed that we can generally mitigate bias without losing accuracy. Thus, future work can be to propose a novel approach to utilize AutoML to mitigate bias for DL systems. While our current approach has shown success in optimizing hyperparameters to resolve fairness bugs in conventional AutoML frameworks, deep learning presents a unique set of challenges due to the complexity and scale of models, which include a higher dimensionality of hyperparameters and more intricate fairness concerns.

These problems can be addressed by adapting the dynamic optimization function to handle the nuances of deep learning architectures. This will involve designing more sophisticated optimization algorithms capable of balancing fairness and accuracy across deeper layers and non-linear interactions, which are characteristic of deep learning models. We can also enhance the search space pruning approach to effectively manage the much larger hyperparameter search spaces typical in deep learning. We believe that by leveraging techniques such as neural architecture search (NAS) and reinforcement learning, we can efficiently navigate this expanded search space to maintain computational efficiency while ensuring fairness improvements. The introduction of new pruning strategies can focus on reducing redundancy in architectural choices and optimizing fairness objectives at different stages of model training.

5.1.2 Fairness Bug Localization for Deep Learning Models

Real-world DL models are often highly complex, involving millions or even billions of parameters, intricate architectures such as deep neural networks with multiple layers, and non-linear transformations. This complexity, while powerful for tasks like image recognition, natural language processing, and decision-making systems, also makes these models opaque and difficult to interpret. As a result, biases present in the training data or the model architecture itself can become embedded within the model in ways that are not immediately obvious. These biases can lead to unfair or discriminatory outcomes when the model is deployed, affecting real-world applications in areas like hiring, loan approval, law enforcement, and healthcare.

Given this complexity, there is a pressing need for robust tools and methods that can effectively localize and diagnose bias issues within deep learning models. Identifying bias in DL models is not straightforward because it can manifest at various stages of the model lifecycle. Bias can originate from biased training data, where certain groups may be underrepresented or misrepresented. It can also arise from the model's design, such as the choice of features, the loss function, or the training process, which may inadvertently favor certain outcomes over others. Furthermore, biases can be amplified during model training and decision-making processes, making it challenging to pinpoint the exact source or nature of the problem. My previous research on fairness specification shows the potential for fairness bug localization for DL models.

5.1.3 Balancing Modular and Global Fairness in Machine Learning Pipelines

One key area for investigation is how modular fairness and global fairness interact with each other. Improvements in fairness at the modular level, such as during data preprocessing or feature selection, do not necessarily guarantee fair outcomes at the global level, as the effects of these improvements may diminish or change when combined with biases introduced by other pipeline components. Conversely, addressing fairness at the global level without paying attention to the modular components may overlook the root causes of the bias, making it difficult to diagnose and correct localized fairness issues. Future research could focus on developing methods to quantify the impact that modular fairness interventions have on the global fairness of the model and vice versa. This would involve tracing fairness issues as they propagate through the pipeline, from individual components to the final model predictions, allowing researchers to understand the complex dependencies between modular and global fairness outcomes.

Bibliography

[1] Giang Nguyen, Sumon Biswas, and Hridesh Rajan. 2023. Fix Fairness, Don't Ruin Accuracy: Performance Aware Fairness Repair using AutoML. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/3611643.3616257