Understanding and reasoning fairness in machine learning pipelines

by

Sumon Biswas

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University

Ames, Iowa

2022

DEDICATION

To my parents who are the biggest source of my motivation, had always belief in me, and encouraged to go on every adventurer, especially this one.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I want to take this opportunity to acknowledge everyone who helped me with different aspects of conducting research and the writing of this dissertation. First, I would like to thank Dr. Hridesh Rajan for his supervision, patience, and 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 would like to thank my committee members Dr. Wei Le, Dr. Andrew Miner, Dr. Simanta Mitra, and Dr. Jia (Kevin) Liu, for their guidance and contributions to this work. I want to extend my thanks to all the Laboratory for Software Design members for offering constructive feedback and timely suggestions during research. Last but not least, my wife and my parents have been highly supportive over all these years. I would not have made this far without their unconditional love and continuous support.

The majority of this draft is adopted from the previous peer-reviewed papers published in top-tier software engineering venues. Chapter 3 is based on our paper published in the proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020) [27]. Chapter 5 is based on the paper published in the proceedings of ESEC/FSE 2021 [29]. Chapter 4 is based on the paper accepted in the International Conference on Software Engineering (ICSE 2022) [31] and a large-scale dataset published in the International Conference on Mining Software Repositories (MSR 2019) [24]. The Chapter 6 is based on the extended version of the paper submitted to ESEC/FSE 2022, which is currently under review. Furthermore, I supported open science as I continuously published code, data, and benchmark as peer reviewed artifacts [28, 26, 23] during writing this dissertation. In this dissertation, I do not include some of the other research works that I conducted during my Ph.D., e.g., propose Auto ML technique to improve fairness, study evolution of deep learning models, analyzing technical debts in ML, and understanding fairness composition in ensemble learning.

ABSTRACT

Machine learning (ML) algorithms are increasingly being used in critical decision making software such as criminal sentencing, hiring employees, approving bank loans, college admission systems, which affect human lives directly. Algorithmic fairness of these ML based software has become a major concern in the recent past. Many incidents have been reported where ML models discriminated people based on their *protected attributes* e.g., race, sex, age, religious belief, etc. Research has been conducted to test and mitigate unfairness in ML models. However, there is a large gap between the theory of ML fairness and how the property can be ensured in practice. Similar to analyzing traditional software defects, fairness has to be engineered in ML software to minimize and eventually guarantee bias-free decisions. In this dissertation, we are the first to introduce compositional reasoning of *group fairness* in ML pipeline and propose *individual fairness* verification technique for neural networks. Towards that goal, first, we conducted a large-scale empirical study to understand unfairness issues in open-source ML models. A number of definitions of algorithmic fairness have been proposed in the literature and many bias mitigation techniques have been proposed. *Group fairness* property ensures that the protected groups (e.g., male-vs-female, young-vs-old, etc.) get similar treatment in the prediction. On the other hand, *individual fairness* states that any two similar individuals should be predicted similarly irrespective of their protected attributes. Often an accuracy-fairness tradeoff is experienced when a mitigation algorithm is applied. We evaluated fairness of models collected from Kaggle and investigated their root causes, compared the performance of mitigation algorithms and their impacts on accuracy. The study suggests many algorithmic components in ML pipeline as the root cause of bias, which leads to our work on compositional reasoning of fairness.

For ML tasks, it is a common practice to build a pipeline that includes an ordered set of stages from acquisition, to preprocessing, to modeling, and so on. However, no research has been conducted to measure fairness of a specific stage or data transformer operators in the pipeline. The existing metrics measure the fairness of the pipeline holistically. We proposed causal reasoning in ML pipeline to measure and instrument fairness of data preprocessing stages. We leveraged existing metrics to define component-specific fairness and localize fairness issues in the pipeline. We also showed how the local fairness of a preprocessing stage composes in the global fairness of the pipeline. In addition, we used the fairness composition to choose appropriate downstream transformer that mitigates unfairness. Although we could identify and localize unfairness in the ML model, providing formal guarantees of fairness is challenging because of the complex decision-making process. Therefore, we proposed *Fairify*, an approach to verify individual fairness property in neural networks (NN). Fairify leverages white-box access and neural pruning to provide certification or counterexample. The key idea is that many neurons in the NN always remain inactive for certain smaller parts of the input domain. So, Fairify applies input partitioning and then prunes the NN for each partition to make them amenable to verification. In this work, we proposed the first SMT-based fairness verification that can answer targeted fairness queries with relaxations as well as provide counterexamples.

Machine learning (ML) algorithms are increasingly being used in critical decision making software such as criminal sentencing, hiring employees, approving bank loans, college admission systems, which affect human lives directly. Algorithmic fairness of these ML based software has become a major concern in the recent past. Many incidents have been reported where ML models discriminated people based on their *protected attributes* e.g., race, sex, age, religious belief, etc. Research has been conducted to test and mitigate unfairness in ML models. However, there is a large gap between the theory of ML fairness and how the property can be ensured in practice. Similar to analyzing traditional software defects, fairness has to be engineered in ML software to minimize and eventually guarantee bias-free decisions. In this dissertation, we are the first to introduce compositional reasoning of *group fairness* in ML pipeline and propose *individual fairness* verification technique for neural networks. Towards that goal, first, we conducted a large-scale empirical study to understand unfairness issues in open-source ML models. A number of definitions of algorithmic fairness have been proposed in the literature and many bias mitigation techniques have been proposed. *Group fairness* property ensures that the protected groups (e.g., male-vs-female, young-vs-old, etc.) get similar treatment in the prediction. On the other hand, *individual fairness* states that any two similar individuals should be predicted similarly irrespective of their protected attributes. Often an accuracy-fairness tradeoff is experienced when a mitigation algorithm is applied. We evaluated fairness of models collected from Kaggle and investigated their root causes, compared the performance of mitigation algorithms and their impacts on accuracy. The study suggests many algorithmic components in ML pipeline as the root cause of bias, which leads to our work on compositional reasoning of fairness.

For ML tasks, it is a common practice to build a pipeline that includes an ordered set of stages from acquisition, to preprocessing, to modeling, and so on. However, no research has been conducted to measure fairness of a specific stage or data transformer operators in the pipeline. The existing metrics measure the fairness of the pipeline holistically. We proposed causal reasoning in ML pipeline to measure and instrument fairness of data preprocessing stages. We leveraged existing metrics to define component-specific fairness and localize fairness issues in the pipeline. We also showed how the local fairness of a preprocessing stage composes in the global fairness of the pipeline. In addition, we used the fairness composition to choose appropriate downstream transformer that mitigates unfairness. Although we could identify and localize unfairness in the ML model, providing formal guarantees of fairness is challenging because of the complex decision-making process. Therefore, we proposed *Fairify*, an approach to verify individual fairness property in neural networks (NN). Fairify leverages white-box access and neural pruning to provide certification or counterexample. The key idea is that many neurons in the NN always remain inactive for certain smaller parts of the input domain. So, Fairify applies input partitioning and then prunes the NN for each partition to make them amenable to verification. In this work, we proposed the first SMT-based fairness verification that can answer targeted fairness queries with relaxations as well as provide counterexamples.

CHAPTER 1. INTRODUCTION

With the rapidly increasing use of machine learning (ML) based software, algorithmic fairness concerns are posing significant risks on human lives and society. This dissertation aims at identifying, assessing, and verifying fairness risks in different stages of ML pipeline e.g., preprocessing, modeling, and make such software more dependable.

ML models are increasingly being used in applications that impact human lives directly such as loan approval, criminal sentencing, hiring employees, etc [54, 136, 10]. Several recent fairness violations manifested the need for research to eliminate algorithmic bias [5, 27, 29, 40, 90, 39, 216, 217]. Although in recent years many research have been conducted to counter unfairness, still software engineers need effective and scalable tools to measure, localize, test, verify, and repair fairness issues in practice [34, 88, 65, 27, 85, 42]. Algorithms have been developed to measure unfairness [210, 56, 64, 84, 36, 48, 212, 181] and mitigate them to a certain extent [213, 36, 210, 48, 73, 84, 148, 114]. Accuracy being the driving factor, often ML models lack proper fairness specification, and analysis methodology that leaves the opportunity to guarantee fairness. The problem is even more challenging because of various fairness metrics, tradeoffs, and the black-box nature of ML. In this dissertation, we unraveled several software engineering approaches for fairness i.e., modular reasoning, localize/repair, verification techniques in ML pipeline that would assist researchers and practitioners to build fairer software.

First, we studied how much fairness issues exist in open-source ML models, what are the available mitigation techniques, and what are their impacts. The empirical evaluation and analyses resulted in several findings. In this work, we analyzed ML models collected from Kaggle and identified several model constructs that affect fairness. Many ML libraries provide configuration for those constructs but they are not documented, which shows opportunity for fairness-aware API design and online monitoring tools. In addition, programmers have limited options of specifying fairness constraints and choosing appropriate metrics. We also demonstrated that oftentimes fairness has a cost of accuracy. However, among the bias mitigation techniques, the ones operate in preprocessing stage retain the accuracy. On the other hand, post-preprocessing mitigation techniques are costly but comparatively more successful in debiasing highly biased models. The work outlined the importance of fairness-aware data sampling and feature selection. The results also showed the fairness effect of data transformers such as scaling, standardizing, etc., which paved the way to my follow-up work.

Existing metrics measure fairness of ML pipelines wholistically based on the outcome [66, 64, 5, 32]. However, the pipelines consist of several components such as encoding, normalization, imputation, feature selection, etc. Our previous work showed that many of those components can directly affect fairness positively or negatively. But existing methods cannot measure fairness of the components, leaving developers unsure of how to repair the fairness issues. Towards achieving that goal, we noticed that, it is essential to understand the nature of the ML pipeline structure. So, we studied data science pipeline as a software architecture, and outlined its characteristics and organization. What are the typical stages (preprocessing, modeling, training, etc.) in data science pipeline and how are they connected? Do the pipelines differ in the theoretical representation and that in the practice? We studied data science pipelines in theory (literature), in the small (standalone notebooks in Kaggle), and in the large (mature DS projects in GitHub). We combined the qualitative and quantitative methodologies to present representative views of DS pipeline in the aforementioned three different sources. The manual investigation by open-coding methodology and the automated pipeline extraction showed comparative analysis of the pipelines.

Then we developed causal reasoning in ML pipeline to measure component-level fairness and quantify fairness of data transformers. Suppose, $\mathcal{P}$ is a canonical pipeline with m preprocessing stages $S_1, S_2, \ldots S_m$ and a final classifier S_m. Our goal was to measure fairness of stage S_k, where $1 \leq k < m$. From pipeline $\mathcal{P}$, we construct another pipeline $\mathcal{P}^*$ by removing/replacing S_k and observe the causal effect on the predictions $\hat{Y}$. Specifically, the component-specific fairness is given by the causal changes from $\hat{Y}(\mathcal{P})$ to $\hat{Y}(\mathcal{P}^*)$. We used four existing fairness criteria e.g., demographic parity, equal opportunity, to measure the causal effect. We also conducted a case study to show how local fairness composes into global fairness. The composition can facilitate developers to choose/replace downstream data transformer that can mitigate unfairness of an upstream one, and make the ML pipeline fairer.

The availability of powerful computational resources and high volume of data has enabled the success of neural networks (NN) in many applications. Although testing and verifying fairness of ML classifiers e.g., decision tree, support vector machine, received much attention recently [190, 5, 66, 97], verification of NN is hard because of the non-linear computational nodes in the network. In this final work, we proposed *Fairify* that verifies individual fairness for NNs in productions. The *individual fairness* property ensures that any two individuals with *similar* attributes except the protected attribute(s) should receive *similar* prediction. We transferred the property as fairness pre- and post-condition of the NN and then used SMT based technique to provide certification or counterexample. Fairify leveraged white-box access to the NN and analyzed activation patterns to provide fairness guarantee. The verification has been made tractable in three steps: 1) Input partitioning, 2) Sound neural pruning, and 3) Heuristic-based neural pruning. The key idea behind the technique is that many neurons remain always inactive and hence do not impact decision making, when we consider only a smaller part of the input domain. Therefore, Fairify divides the input space into smaller partitions and assigns a copy of NN to each partition. Then Fairify employs interval arithmetic and layer-wise verification on each neuron to perform sound neural pruning. When the sound pruning is not sufficient, Fairify applies the heuristics of neural activation to prune further. The developer may choose to deploy the pruned (and certified) version of the NN with small lose of accuracy. The technique is scalable to real-world NNs as well as to the relaxed version of fairness queries. Finally, Fairify can provide partial certification and locate fairness violations with counterexample, which would make fairness repair explainable and interactive.

1.1 Contribution

In this dissertation, first, we investigate the fairness issues existing in real-world machine learning models. We identify several software engineering aspects of fairness and opportunities to mitigate bias. The study reinstates that much of the issues remain in the data and data preprocessing has a significant impact on the overall fairness. However, we do not know which of the preprocessing operator is (un)fair. As a solution to the problem, we proposed causal reasoning in the ML pipeline to measure fairness of data transformer components. Another challenge identified in our study is that although many mitigation algorithm and testing strategies have been proposed in the literature, it is hard to provide formal guarantees of fairness in complex ML models. We are the first to propose individual fairness verification technique for neural networks.

In Chapter 3, we presented the empirical study conducted on the fairness of real-world models [27]. No prior work has been done to understand the fairness issues exist on the models in practice and how the existing mitigation techniques perform. We evaluated fairness with respect to a number of metrics and analyzed the root cause of the bias. The results show opportunity to several model constructs in the ML library that impact fairness. We also show opportunities to build automated tools to localize fairness defects in ML software. We also presented the optimization and trade-off problems in fairness, which outlines many future work in the direction. Furthermore, we created a novel benchmark of 40 ML models for 5 fairness aware tasks, which are collected from Kaggle. The benchmark, code, and data are shared in our GitHub repository[1] and also published as an artifact [28].

Then we proposed a novel method leveraging causal reasoning to measure fairness of a component in ML pipeline. Towards that goal, in Chapter 4, we conducted a comprehensive study on data science pipelines to understand their representative structure in theory and practice [31]. Although many prior works used this specific kind of software architecture, called the pipeline, we do not fully understand its organization and characteristics. Therefore, we collected and analyzed a number of pipelines *in theory*, *in-the-small*, and *in-the-large*. The resulting software architecture helped us to formalize the definition of fairness of a stages in ML pipeline. We also published the code, collected pipelines, and analysis results in the artifact[2] so that it can be used in the future work in the area of design and architecture of pipelines [23].

In Chapter 5, we proposed the causal method to quantify fairness of preprocessing stage or each data transformer. Thus, we can identify or localize unfair data preprocessing and instrument those for fairer results. We developed four novel component-specific metrics that can operate on ML pipelines to measure algorithmic bias. Finally, we showed how *local* fairness of the components compose in the *global* fairness of the pipeline. The pipeline benchmark and implementation of new metrics are shared in our artifact[3] [26].

[1]https://github.com/sumonbis/ML-Fairness
[2]https://github.com/sumonbis/DS-Pipeline
[3]https://github.com/sumonbis/FairPreprocessing

Finally, in Chapter 6, we proposed *Fairify*, a technique to verify individual fairness property in neural networks. We are the first to described individual fairness property for neural network as first-order formulas and proposed automated verification. Fairify uses off-the-self constraint solver to provide certification or show counterexample. We leveraged input partitioning technique and compositional verification technique so that we could verify various relaxation of individual fairness property. In addition, Fairify can provide partial verification that gives the developers ability to verify targeted fairness queries. A new neural network benchmark for fairness property as well as the implementation of our tool is shared in the Github repository[4].

1.2 Outline

The rest of the dissertation is organized as follows. Chapter 2 describes the related works. Chapter 3 presents the empirical study conducted on the fairness issues of real-world machine learning models. The findings in the study inspired our next work on compositional fairness in ML pipeline. Towards that goal, in Chapter 4, we present a comprehensive study on the representative structure of data science pipeline. In Chapter 5, we propose the causal method to isolate a component of ML pipeline and measure its fairness. We describe the methods to identify unfair preprocessing stages in the pipeline and show their composition. In Chapter 6, we present a verification technique called Fairify to provide formal guarantee of individual fairness property. Finally, in Chapter 7, we discuss future works and conclude.

[4]https://github.com/anonymous-authorss/Fairify

CHAPTER 2. RELATED WORKS

2.1 Fairness in ML Classification

The machine learning community has defined many fairness criteria and proposed metrics to measure the fairness of classification tasks [210, 54, 145, 56, 64, 84, 36, 48, 212, 181]. Different metrics are appropriate for certain situations and account for specific perspective of fairness. The metrics are broadly categorized intro two: 1) group fairness and 2) individual fairness. *Group fairness* property ensures that the protected groups (e.g., male-vs-female, young-vs-old, etc.) get similar treatment in the prediction of the model. On the other hand, *individual fairness* states that any two similar individuals who differ only in their protected attribute get similar treatment in the model prediction [66, 97]. Galhotra et al. argued that group fairness property might not detect bias in scenarios when same amount of discrimination is made for any two groups [66], which led to the usage of individual fairness property in many recent works [66, 217, 5, 190, 218]. In this dissertation, we considered both group and individual fairness criteria as appropriate. Especially, we adopted the most used fairness metrics from the literature.

Following the measures of fairness in ML models, many mitigation techniques have also been proposed to eliminate bias from prediction [113, 213, 64, 54, 36, 210, 48, 73, 115, 84, 148, 114]. These techniques can be broadly classified into preprocessing, in-processing, and postprocessing approaches. *Preprocessing algorithms* such as Reweighing [113], Disparate Impact Remover [64], do not change the model (e.g., algorithm, hyperparameters, structure, training) and only work on the dataset before training so that models can produce fairer predictions. In-processing algorithms such as Adversarial Debiasing [213], Prejudice Remover Regularizer [115], modify the ML models to mitigate the bias in the original model predictions. The post-processing algorithms such as Equalized Odds [84], Calibrated Equalized Odds [148], Reject Option Classification [114], modify the prediction result instead of the ML models or the training data. Because of the numerous usage and their impacts, the above works focus on the fairness of classification tasks. This dissertation also concentrates on ML classification tasks. Therefore, fairness of ML regression or recommendation systems are out of the scope, which need need different definition of fairness. Different fairness measures and mitigation algorithms have been discussed in Section 3.3.3 and Section 3.3.6.

2.2 Software Engineering for ML Fairness

Recently, software engineering (SE) community has also focused on the fairness in ML, mostly on fairness testing [188, 66, 190, 5]. 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 build automated tools [3, 190, 179] and libraries [19] for fairness. In addition, empirical studies have been conducted to compare, contrast between fairness aspects, interventions, tradeoffs, developers concerns, and human aspects of fairness [65, 27, 85, 88, 216].

Fairness Testing in ML. FairTest [188] proposes methodology to detect unwarranted feature associations and potential biases in a dataset using manually written tests. Themis [66] generates random tests automatically to detect causal fairness using black-box decision making process. Aequitas [190] is a fully automated directed test generation module to generate discriminatory inputs in ML models, which can be used to validate individual fairness. FairML [3] introduces an orthogonal transformation methodology to quantify the relative dependence of black-box models to its input features, with the goal of assessing fairness. A more recent work [5] proposes black-box fairness testing method to detect individual discrimination in ML models. They [5] propose a test case generation algorithm based on symbolic execution and local explainability. The above works have proposed novel techniques to detect and test fairness in ML systems. However, we have focused on understanding other software engineering aspects such as understanding tradeoffs, compositional reasoning, and providing provable guarantee.

Empirical Study on ML Fairness. Friedler *et al.* worked on an empirical study on fairness but compared the performance of fairness enhancing interventions and not the ML models [65]. Harrison *et al.* conducted a survey based empirical study to understand how fairness of different models is perceived by humans [85]. Holstein *et al.* also conducted survey on industry developers to find the challenges for developing fairness-aware tools and models [88]. However, no empirical study has been conducted to measure and compare fairness of ML models in practice, and analyze the impacts of mitigation algorithms on the models. We are the first to create real-world fairness benchmark of ML models and investigate a comprehensive set of fairness metrics, mitigation algorithms and their impacts.

2.3 ML Pipeline and Composition of Stages

However, real-world machine learning software operate in a complex environment [32, 51]. In an ML task, the prediction is made after going through a series of stages such as data cleaning, feature engineering, etc., which build the machine learning pipeline [8, 207]. Studying only the fairness of the classifiers (e.g., *Decision Tree*, *Logistic Regression*) fails to capture the fairness impact made by other stages in ML pipeline. In this dissertation, we conducted a detailed analysis of data science pipelines and their structure (stages, connections, feedback loops) in theory and practice. The study led us to analyze fairness of each stage in the pipeline and provide compositional reasoning.

Data Science Pipeline. The term *pipeline* was introduced by Garlan with *box-and-line* diagrams and explanatory prose that assist software developers to design and describe complex systems so that the software becomes intelligible [70]. Shaw and Garlan have provided the *pipes-and-filter* design pattern that involves stages with processing units (filters) and ordered connections (pipes) [175]. They also argued that pipeline gives proper semantics and vocabulary which helps to describe the concerns, constraints, relationship between the sub-systems, and overall computational paradigm [70, 175]. By *data science pipeline* (DS pipeline), we are referring to a series of processing *stage*s that interact with data, usually acquisition, management, analysis, and reasoning [137, 135]. DS pipeline is an umbrella term incorporate ML pipelines and other data pipelines (described in Chapter 4).

ML Pipelines Many studies presented ML pipeline in their own context, which can not be generalized across all DS systems. Garcia et al. focused on building an iterative process with three main phases: development, training and inference. They described the interpretation of data and code while integrating the whole lifecycle [69]. Polyzotis et al. presented the challenges of data management in building production-level ML pipeline in Google around three broad themes: data understanding, data validation and cleaning, and data preparation [149, 150]. They also provided an overview of an end-to-end large-scale ML pipeline with a data point of view. Carlton E. Sapp defined ML concepts, business challenges, stages in the lifecycle, roles of DS teams with comprehensive end-to-end ML architecture [165]. This gives us a holistic understanding of the business processes (e.g., acquire, organize, analyze, deliver) of a DS project.

A few other studies try to capture the DS process by surveying and interviewing developers. Roh et al. surveyed the data collection techniques in the field of big data. They presented the workflow of data collection answering how to improve data or models in an ML system [156]. Another study identified the software engineering practices and challenges in building AI applications inside Microsoft development teams [9]. They found some key differences in AI software process compared to other domains. They considered a 9-stage workflow for DS software development. Hill et al. interviewed experienced AI developers and identified problems they face in each stage [86]. They also tried to compare the traditional software process and the AI process. Zhou presented her own view to build a better ML pipeline [220]. They presented three challenges in building ML pipelines: data quality, reliability and accessibility.

Some articles described ML applications and frameworks which present DS pipelines from industry. For example, *Databricks* provides high-level APIs for programming languages [89]. Team Data Science Process (TDSP) is an agile and iterative process to build intelligent applications inside Microsoft corporation [174]. In a US patent, the authors compared two data analytic lifecycles [187], and presented the difference in the set of parameters with respect to time and cost. CRoss Industry Standard Process for Data Mining (CRISP-DM) is a 6-stage comprehensive process model for data mining projects across any industry [204]. Google Cloud Blog described the workflow of an AI platform [76]. They explained tasks completed in each stage with respect to Google Cloud and TensorFlow[1]. Although there are many papers in the literature presenting DS pipeline, there is no comprehensive study that tries to understand and compare DS pipelines in theory and practice.

Fairness Composition in ML pipeline Dwork and Ilvento argued that fairness is dynamic in a multi-component environment [57]. The authors studied how fairness composes in a multi-classification setting. Specifically, they proposed the theory of "functional composition", where the binary output of multiple classifiers are combined through logical operators such as AND, OR, etc. Bower *et al.* discussed fairness in ML *pipeline*, where they considered *pipeline* as sequence of multiple classification tasks [32]. They also showed that when decisions of fair components are compounded, the final decision might not be fair. They demonstrated the fairness propagation using equal opportunity definition of fairness and a two-stage hiring pipeline example, wherein each stage the candidates are selected for the next stage. Grgic-Hlaca *et al.* discussed procedural fairness with respect to different features in a dataset [81]. D'Amour *et al.* studied the dynamics of fairness in multi-classification environnement using simulation [51]. Recently, Yang *et al.* [207] proposed software instrumentation in ML pipeline and [211] proposed a methodology to detect impact of a preprocessing stage in ML pipeline.

2.4 Testing and Verification of ML Models

ML Model Testing. DeepCheck [79] proposes lightweight white-box symbolic analysis to validate deep neural networks (DNN). DeepXplore [147] proposes a white-box framework to generate test input that can exploit the incorrect behavior of DNNs. DeepTest [186] uses domain-specific metamorphic relations to detect errors in DNN based software. These works have focused on the robustness property of ML systems, whereas we have studied fairness property that is fundamentally different from robustness [190].

Causal Reasoning in ML. The causality theorem was proposed by Pearl [144, 145] and further studied extensively to reason about fairness in many classification scenarios [66, 127, 215, 163, 160]. Causality notion of fairness captures that everything else being equal, the prediction would not be changed in the counterfactual world where only an intervention happens on a variable [66, 127, 160]. For example, Galhotra et al. proposed causal discrimination score for fairness testing [66]. The authors created test inputs by altering original protected attribute values of each data instance, and observed whether prediction is changed for those test inputs. This causal reasoning of fairness is a stronger notion since it provides causality in software by observing changes in the outcome made by a specific stage in the pipeline [66, 145].

NN Verification for Different Properties. The robustness of NN has gained a lot of attention for safety-critical applications e.g., autonomous vehicles [134, 44, 75]. Algorithms have been proposed to detect adversarial inputs and satisfy local robustness property [13, 37, 63, 91, 117]. Robustness verification has been proposed using SMT based techniques [92, 78, 16]. Also, verification algorithms have been proposed that uses off-the-shelve SMT solver [61, 92]. Research has also been conducted to compute tight bounds of the neurons and provide probabilistic guarantees for some properties [183, 199, 178]. With the extensive use of NN, many recent work focused on new types of properties and both static and dynamic analysis [143, 37, 130, 186, 93]. Xiao et al. showed that the decision of early layers can be used to give warning [206]. Finally, many prior works focused on other dynamic techniques and testing methodologies to suggest violation of safety properties [37, 130, 186, 93].

Fairness Verification. A few recent works proposed individual fairness testing on NN [218, 217]. While input test generation has been helpful to find fairness violations, verification is more difficult since it proves certain property for all possible inputs. Probabilistic verification techniques have been proposed to verify group fairness property [6, 17]. Albarghouthi et al. proposed FairSquare to verify group fairness as a probabilistic property [6]. Bastani et al. also proposed scalable technique to verify group fairness for ML and NN models [17]. Recently, John et al. proposed individual fairness verification approach for three different kinds of ML models [97]. Urban et al. proposed Libra to provide dependency fairness certification for NN using abstract interpretation [191]. They used forward and backward analysis to compute input region that is fair or unfair using different abstract domains e.g., Boxes, Symbolic, Deeppoly. Mazzucato and Urban extended Libra with another abstract domain (Neurify) and implemented automated configuration for scalability and parallelization [131]. Dependency fairness is a notion close to individual fairness but does not consider similarity relaxation of individuals. In addition, Libra does not provide counterexample if there is a fairness violation. In this paper, we proposed SMT based verification, which provides certification or counterexample for the NN as well it outputs pruned NN that has added benefit.

CHAPTER 3. DO THE MACHINE LEARNING MODELS ON A CROWD SOURCED PLATFORM EXHIBIT BIAS? AND EMPIRICAL STUDY ON MODEL FAIRNESS

Machine learning models are used in important decision-making software such as approving bank loans, recommending criminal sentencing, hiring employees, etc. It is important to ensure the fairness of these models so that no discrimination is made based on *protected attribute* (e.g., race, sex, age) while decision making. Algorithms have been developed to measure unfairness and mitigate them to a certain extent. In this chapter, we focused on the empirical evaluation of fairness and mitigations on real-world machine learning models. We found that some model optimization techniques result in inducing unfairness in the models. On the other hand, although there are some fairness control mechanisms in machine learning libraries, they are not documented. The mitigation algorithm also exhibit common patterns such as mitigation in the post-processing is often costly (in terms of performance) and mitigation in the preprocessing stage is preferred in most cases. We also presented different trade-off choices of fairness mitigation decisions. Our study suggests future research directions to reduce the gap between theoretical fairness aware algorithms and the software engineering methods to leverage them in practice.

3.1 Introduction

Since machine learning (ML) models are increasingly being used in making important decisions that affect human lives, it is important to ensure that the prediction is not biased toward any protected attribute such as race, sex, age, marital status, etc. ML fairness has been studied for about past 10 years [65], and several fairness metrics and mitigation techniques [113, 64, 210, 36, 48, 213, 115, 84, 213] have been proposed. Many testing strategies have been developed [190, 5, 66] to detect unfairness in software systems. Recently, a few tools have been proposed [4, 19, 188, 179] to enhance fairness of ML classifiers. However, we are not aware how much fairness issues exist in ML models from practice. Do the models exhibit bias? If yes, what are the different bias types and what are the model constructs related to the bias? Also, is there a pattern of fairness measures when different mitigation algorithms are applied? In this chapter, we conducted an empirical study on ML models to understand these characteristics.

Harrison *et al.* studied how ML model fairness is perceived by 502 Mechanical Turk workers [85]. Recently, Holstein *et al.* conducted an empirical study on ML fairness by surveying and interviewing industry practitioners [88]. They outlined the challenges faced by the developers and the support they need to build fair ML systems. They also discussed that it is important to understand the fairness of existing ML models and improve software engineering to achieve fairness. In this chapter, we analyzed the fairness of 40 ML models collected from a crowd sourced platform, Kaggle, and answered the following research questions.

- **RQ1: (Unfairness)** What are the unfairness measures of the ML models in the wild, and which of them are more or less prone to bias?
- **RQ2: (Bias mitigation)** What are the root causes of the bias in ML models, and what kind of techniques can successfully mitigate those bias?
- **RQ3: (Impact)** What are the impacts of applying different bias mitigating techniques on ML models?

First, We created a benchmark of 40 top-rated models from Kaggle used for 5 different task. We manually verified the models and selected appropriate ones for the analysis. Then using a comprehensive set of fairness metrics, evaluated their fairness. Then, we applied 7 mitigation techniques on these models and analyzed the fairness, mitigation results, and impacts on performance. Finally, we analyzed the results to answer the above research questions. The key findings are: model optimization goals are configured towards overall performance improvement, causing unfairness. A few model constructs are directly related to fairness of the model. However, ML libraries do not explicitly mention fairness in documentation. Models with effective pre-processing mitigation algorithm are more reliable and pre-processing mitigations always retain performance. We also reported different patterns of exhibiting bias and mitigating them. Finally, we reported the trade-off concerns evident for those models.

The chapter is organized as follows: Section 3.2 describes the background of fairness in machine learning models. In Section 3.3, we described the methodology of creating the benchmark and setting up experiment, and discussed the fairness metrics and mitigation techniques. Section 3.4 describes the fairness comparison of the models, Section 3.5 describes the mitigation techniques, and Section 3.6 describes the impacts of mitigation. Finally, we discussed the threats to validity and future work in Section 3.7.

3.2 Background

The basic idea of ML fairness is that the model should not discriminate between different individuals or groups from the protected attribute class [65, 66]. *Protected attribute* (e.g., race, sex, age, religion) is an input feature, which should not affect the decision making of the models solely. Chen *et al.* listed 12 protected attributes for fairness analysis [46]. One trivial idea is to remove the protected attribute from the dataset and use that as training data. Pedreshi *et al.* showed that due to the redundant encoding of training data, it is possible that protected attribute is propagated to other correlated attributes [146]. Therefore, we need fairness aware algorithms to avoid bias in ML models. In this paper, we have considered both group fairness and individual fairness. *Group fairness* measures whether the model prediction discriminates between different groups in the protected attribute class (e.g., sex: *male/female*) [56]. *Individual fairness* measures whether similar prediction is made for similar individuals those are only different in protected attribute [56]. Based on different definitions of fairness, many group and individual fairness metrics have been proposed. Additionally, many fairness mitigation techniques have been developed to remove unfairness or bias from the model prediction. The fairness metrics and mitigation techniques have been described in the next section.

3.3 Methodology

In this section, first, we described the methodology to create the benchmark of ML models for fairness analysis. Then we described our experiment design and setup. Finally, we discussed the fairness metrics we evaluated and mitigation algorithms we applied on each model.

3.3.1 Benchmark Collection

We collected ML models from Kaggle kernels [99]. Kaggle is one of the most popular data science (DS) platform owned by Google. Data scientists, researchers, and developers can host or take part in DS competition, share dataset, task, and solution. Many Kaggle solutions resulted in impactful ML algorithms

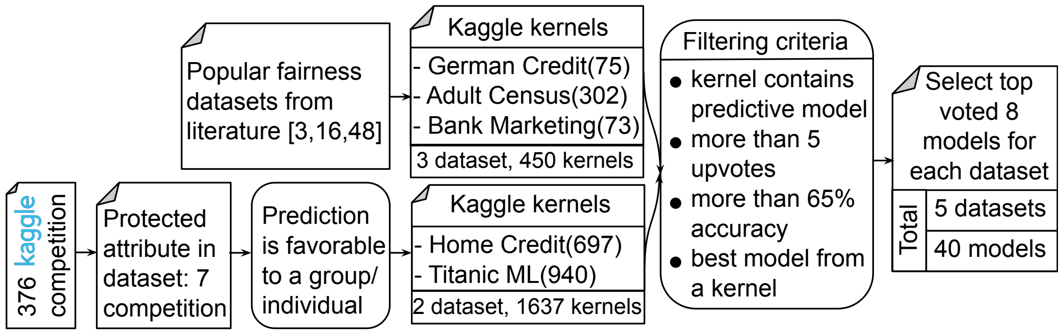

Figure 3.1: Benchmark model collection process

and research such as neural networks used by Geoffrey Hinton and George Dahl [50], improving the search for the Higgs Boson at CERN [96], state-of-the-art HIV research [38], etc. There are 376 competitions and 28,622 datasets in Kaggle to date. The users can submit solutions for the competitions and dataset-specific tasks. To create a benchmark to analyze the fairness of ML models, we collected 40 kernels from the Kaggle. Each kernel provides solution (code and description) for a specific data science task. In this study, we analyzed ML models that operate on 1) datasets utilized by prior studies on fairness, and 2) datasets with protected attribute (e.g., sex, race). With this goal, we collected the ML models with different filtering criteria for each category. The overall process of collecting the benchmark has been depicted in Figure 3.1.

To identify the datasets used in prior fairness studies, we refer to the work on fairness testing by Galhotra *et al.* [66], where two datasets, German Credit and Adult Census have been used. Udeshi *et al.* experimented on models for the Adult Census dataset [190]. Aggarwal *et al.* used six datasets: German Credit, Adult Census, Bank Marketing, US Executions, Fraud Detection, and Raw Car Rentals) [5]. Among these datasets, German Credit, Adult Census and Bank Marketing dataset are available on Kaggle. From the solutions for these datasets, we collected 440 kernels (65 for German Credit, 302 for Adult Census, and 73 for Bank Marketing). Furthermore, we filtered the kernels based on three criteria to select the top-rated ones: 1) contain predictive models (some kernels only contain exploratory data analysis), 2) at least 5 upvotes, and 3) accuracy $\geq 65\%$. Often a kernel contains multiple models and tries to find the best performing one. In these cases, we selected the best performing model from every kernel. Thus, we selected the top 8 models based on upvotes for each of the 3 datasets and got 24 ML models.

Table 3.1: The datasets used in the fairness experimentation. # F: Feature count. PA: Protected attribute.

Dataset	**Size**	**# F**	**PA**	**Description**
German Credit [103]	1,000	21	age, sex	This dataset contains personal information about individuals and predicts credit risk (good or bad credit). The *age* protected attribute is categorized into young (< 25) and old (≥ 25) based on [65].
Adult Census [100]	32,561	12	race, sex	This dataset comprises of individual information from the 1994 U.S. census. The target feature of this dataset is to predict whether an individual earns $\geq \$50,000$ or not in a year.
Bank Marketing [101]	41,188	20	age	This dataset contains the direct marketing campaigns data of a Portuguese bank. The goal is to predict whether a client will subscribe for a term deposit or not.
Home Credit [104]	3,075,11	240	sex	This dataset contains data related to loan applications for individuals who do not get loan from the traditional banks. The target feature is to predict whether an individual who can repay the loan, get the application accepted or not.
Titanic ML [105]	891	10	sex	This dataset contains data about the passengers of Titanic. The target feature is to predict whether the passenger survived the sinking of Titanic or not. The target of the test set is not published. So, we took the training data and further split it into train and test.

Chen *et al.* [46] listed 12 protected attributes, e.g., age, sex, race, etc. for fairness analysis. We found 7 competitions in Kaggle, that contain any of these attributes. From the selected ones, we filtered out the competitions that involve prediction decisions not being favorable to individuals or a specific group. For example, although this competition [102] has customers *age* and *sex* in the dataset, the classification task is to recommend an appropriate product to the customers, which we can not classify as fair or unfair. Thus, we got two appropriate competitions with several kernels. To select ML models from these competitions, we utilized the same filtering criteria used before and selected 8 models for each dataset based on the upvotes. Finally, we created a benchmark containing 40 top-rated Kaggle models that operate on 5 datasets. The characteristics of the datasets and tasks in the benchmark are shown in Table 3.1.

3.3.2 Experiment Design

After creating the benchmark, we experimented on the models, evaluated performance and fairness metrics, and applied different bias mitigation techniques to observe the impacts. Our experiment design process is shown in Figure 3.2. The experiments on the benchmark have been peer reviewed in ESEC/FSE 2020 and published as an artifact [28]. All the source code, datasets, benchmark, and experiemnt details can be found in our GitHub repository[1], which is also peer reviewed and published as an artifact [28].

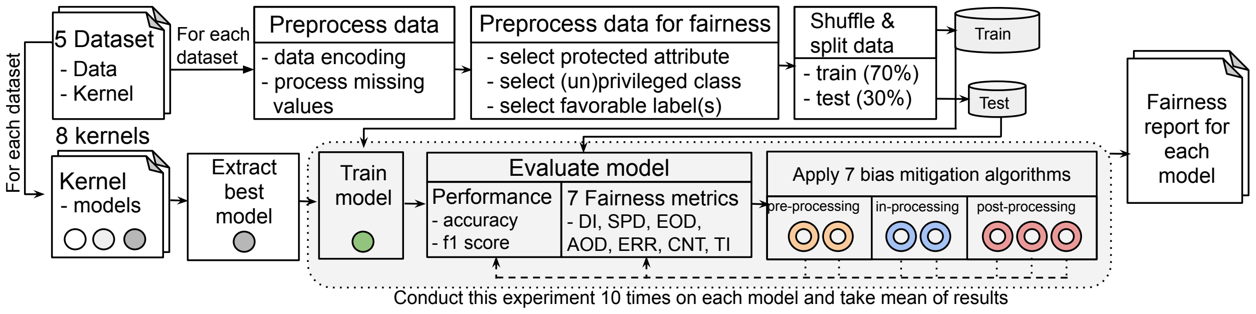

Figure 3.2: Experimentation to compute performance, fairness and mitigation impacts of machine learning models

In our benchmark, we have models from five dataset categories. To be able to compare the fairness of different models in each dataset category, we used the same data preprocessing strategy. We processed the missing or invalid values, transformed continuous features to categorical (e.g., age$<$25: young, age$\geq$25: old), and converted non-numerical features to numerical (e.g., *female*: 0, *male*: 1). We have done some further preprocessing to the dataset to be used for fairness analysis: specify the protected attributes, privileged and unprivileged group, and what are the favorable label or outcome of the prediction. For example, in the Home Credit dataset, *sex* is the protected attribute, where *male* is the privileged group, *female* is the unprivileged group, and the prediction label is credit risk of the person i.e., good (favorable label) or bad. For all the datasets, we used shuffling and same train-test splitting (70%-30%) before feeding the data to the models.

For each dataset category, we have eight Kaggle kernels. The kernels contain solution code written in Python for solving classification problems. In general, the kernels follow these stages: data exploration, preprocessing, feature selection, modeling, training, evaluation, and prediction. From the kernels, we have

[1]https://github.com/sumonbis/ML-Fairness

manually extracted the code for modeling, training, and evaluation. For example, this kernel [108] loads the German Credit dataset, performs exploratory analysis and selects a subset of the features for training, preprocesses data, and finally implements XGBoost classifier for predicting the credit risk of individuals. We have manually sliced the code for modeling, training, and evaluation. Often the kernels try multiple models, evaluate results, and find the best model. From a single kernel, we have only sliced the best performing model found by the kernel. Some kernels do not specify the best model. In this case, we selected the model with the best accuracy. For example, this kernel [107] works on Adult Census dataset and implements four models (Logistic Regression, Decision Tree, K-Nearest Neighbor and Gradient Boosting) for predicting income of individuals. We selected the Gradient Boosting classifier model since it gives the best accuracy.

After extracting the best model, we train the model and evaluate performance (accuracy, F1 score). We found that the model performance in our experiment is consistent with the prediction made in the kernel. Then, we evaluated 7 different fairness metrics described in Section 3.3.5. Next, we applied 7 different bias mitigation algorithms separately and evaluated the performance and fairness metrics. Thus, we collect the result of 9 metrics (2 performance metric, 7 fairness metric) before applying any mitigation algorithm and after applying each mitigation algorithm. For each model, we have done this experiment 10 times and taken the mean of the results as suggested by [65]. We used the open-source Python library AIF 360 [19] developed by IBM for fairness metrics and bias mitigation algorithms. All experiments have been executed on a MAC OS 10.15.2, having 4.2 GHz Intel Core i7 processor with 32 GB RAM and Python 3.7.6.

3.3.3 Measures

We computed the algorithmic fairness of each subject model in our benchmark. Let, $D = (X, Y, Z)$ be a dataset where X is the training data, Y is the binary classification label ($Y = 1$ if the label is favorable, otherwise $Y = 0$), Z is the protected attribute ($Z = 1$ for privileged group, otherwise $Z = 0$), and $\hat{Y}$ is the prediction label (1 for favorable decision and 0 for unfavorable decision). If there are multiple groups for protected attributes, we employed a binary grouping strategy (e.g., race attribute in Adult Census dataset has been changed to white/non-white).

3.3.4 Accuracy Measure

Before measuring the fairness of the model, we computed the performance in terms of accuracy, and F1 score.

Accuracy: Accuracy is given by the ratio of truly classified items and total number of items.

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

F1 Score: This metric is given by the harmonic mean of precision and recall.

$$\text{F1} = 2 * (\text{Precision} * \text{Recall})/(\text{Precision} + \text{Recall})$$

3.3.5 Fairness Measure

Many quantitative fairness metrics have been proposed in the literature [22] based on different definitions of fairness. For example, AIF 360 toolkit has APIs for computing 71 fairness metrics [19]. In this paper, without being exhaustive, a representative list of metrics have been selected to evaluate the fairness of ML models. We adopted the metrics recommendation of Friedler *et al.* [65] and further added the individual fairness metrics.

Metrics based on base rates:

Disparate Impact (DI): This metric is given by the ratio between the probability of unprivileged group gets favorable prediction and the probability of privileged group gets favorable prediction [64, 210].

$$\text{DI} = \mathsf{P}[\hat{Y} = 1|Z = 0]/\mathsf{P}[\hat{Y} = 1|Z = 1]$$

Statistical Parity Difference (SPD): This measure is similar to DI but instead of the ratio of probabilities, difference is calculated [36].

$$\text{SPD} = \mathsf{P}[\hat{Y} = 1|Z = 0] - \mathsf{P}[\hat{Y} = 1|Z = 1]$$

Metrics based on group conditioned rates:

Equal Opportunity Difference (EOD): This is given by the true-positive rate (TPR) difference between unprivileged and privileged groups.

$$\text{TPR}_u = \mathsf{P}[\hat{Y} = 1 | Y = 1, Z = 0] \text{ ; } \text{TPR}_p = \mathsf{P}[\hat{Y} = 1 | Y = 1, Z = 1]$$

$$\text{EOD} = TPR_u - TPR_p$$

Average Odds Difference (AOD): This is given by the average of false-positive rate (FPR) difference and true-positive rate difference between unprivileged and privileged groups [84].

$$\text{FPR}_u = \mathsf{P}[\hat{Y} = 1 | Y = 0, Z = 0] \text{ ; } \text{FPR}_p = \mathsf{P}[\hat{Y} = 1 | Y = 0, Z = 1]$$

$$\text{AOD} = \frac{1}{2}\{(FPR_u - FPR_p) + (TPR_u - TPR_p)\}$$

Error Rate Difference (ERD): Error rate is given by the addition of false-positive rate (FPR) and false-negative rate (FNR) [48].

$$\text{ERR} = \text{FPR} + \text{FNR}$$

$$\text{ERD} = \text{ERR}_u - \text{ERR}_p$$

Metrics based on individual fairness:

Consistency (CNT): This individual fairness metric measures how similar the predictions are when the instances are similar [212]. Here, $n\_neighbors$ is the number of neighbors for the KNN algorithm.

$$\text{CNT} = 1 - \frac{1}{n * n\_neighbors} \sum_{i=1}^{n} |\hat{y}_i - \sum_{j \in \mathcal{N}_{n\_neighbors(x_i)}} \hat{y}_j|$$

Theil Index (TI): This metric is also called the entropy index which measures both the group and individual fairness [181]. Theil index is given by the following equation where $b_i = \hat{y}_i - y_i + 1$.

$$\text{TI} = \frac{1}{n} \sum_{i=1}^{n} \frac{b_i}{\mu} \ln \frac{b_i}{\mu}$$

3.3.6 Bias Mitigation Techniques

In this section, we discussed the bias mitigation techniques that have been applied to the models. These techniques can be broadly classified into preprocessing, in-processing, and postprocessing approaches.

3.3.7 Preprocessing Algorithms

Preprocessing algorithms do not change the model and only work on the dataset before training so that models can produce fairer predictions.

Reweighing [113]: In a biased dataset, different weights are assigned to reduce the effect of favoritism of a specific group. If a class of input has been favored, then a lower weight is assigned in comparison to the class not been favored.

Disparate Impact Remover [64]: This algorithm is based on the concept of the metric DI that measures the fraction of individuals achieves positive outcomes from an unprivileged group in comparison to the privileged group. To remove the bias, this technique modifies the value of protected attribute to remove distinguishing factors.

3.3.8 In-processing Algorithms

In-processing algorithms modify the ML model to mitigate the bias in the original model prediction.

Adversarial Debiasing [213]: This approach modifies the ML model by introducing backward feedback (negative gradient) for predicting the protected attribute. This is achieved by incorporating an adversarial model that learns the difference between protected and other attributes that can be utilized to mitigate the bias.

Prejudice Remover Regularizer [115]: If an ML model relies on the decision based on the protected attribute, we call that direct prejudice. In order to remove that, one could simply remove the protected attribute or regulate the effect in the ML model. This technique applies the latter approach, where a regularizer is implemented that computes the effect of the protected attribute.

3.3.9 Post-processing Algorithms

This genre of techniques modifies the prediction result instead of the ML models or the input data.

Equalized Odds (E) [84]: This approach changes the output labels to optimize the EOD metric. In this approach, a linear program is solved to obtain the probabilities of modifying prediction.

Calibrated Equalized Odds [148]: To achieve fairness, this technique also optimizes EOD metric by using the calibrated prediction score produced by the classifier.

Reject Option Classification [114]: This technique favors the instances in privileged group over unprivileged ones that lie in the decision boundary with high uncertainty.

3.4 Unfairness in ML Models

In this section, we explored the answer of RQ1 by analyzing different fairness measures exhibited by the ML models in our benchmark. Do the models have bias in their prediction? If so, which models are fairer and which are more biased? What is causing the models to be more prone to bias? What kind of fairness metric is sensitive to different models? To answer these questions, we conducted experiment on the ML models and computed the fairness metrics. The result is presented in Table 3.2. The unfairness measures for all the 40 models are depicted in Figure 3.3. To be able to compare all the metrics in the same chart, disparate impact (DI), and consistency (CNT) have been plotted in the log scale. If the value of a fairness metric is 0, there is no bias in the model according to the corresponding metric. If the measure is less than or greater than 0, bias exists. The negative bias denotes that the prediction is biased towards privileged group and positive bias denotes that prediction is biased towards unprivileged group.

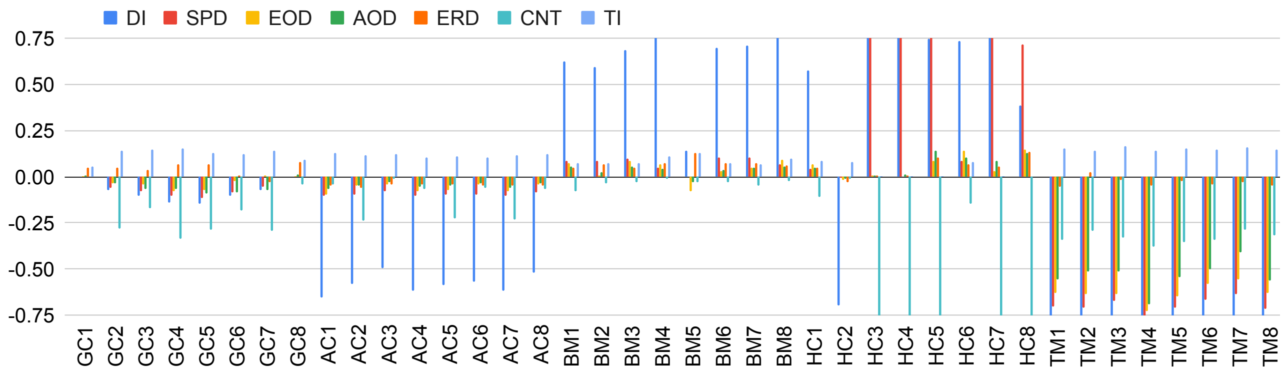

Figure 3.3: Unfairness exhibited by the ML models with respect to different metrics

We found that all the models exhibit unfairness and models specific to a dataset show similar bias patterns. From Figure 3.3, we can see that all the models exhibit bias with respect to most of the fairness metrics. For a model, metric values vary since the metrics follow different definitions of fairness. Therefore, we compared bias of different models both cumulatively and using the specific metric individually. To compare total bias across all the metrics, we have taken the absolute value of the measures and computed the sum of bias for each model. In Figure 3.4, we can see the total bias exhibited by the models. Although the bias exhibited by models for each dataset follow similar pattern, certain models are fairer than others.

Finding 1: Model optimization goals seek overall performance improvement, which is causing unfairness.

Model GC1 exhibits the lowest bias among German Credit models. GC1 is a Random Forest (RFT) classifier model, which is built by using a grid search over a given range of hyperparameters. After the grid search, the best found classifier is:

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None, criterion='gini', max_depth=3, max_features=4,
        max_leaf_nodes=None, max_samples=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1,
        min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=25, n_jobs=None, oob_score=False, random_state=2,
        warm_start=False)
```

We found that GC6 is also a Random Forest classifier built through grid search. However, GC6 is less fair in terms of cumulative bias (Figure 3.4), and individual metrics (Figure 3.3) except error rate difference (ERD). We investigated the reason of the fairness differences in these two models by running both of them by changing one hyperparameter at a time. We found that the fairness difference is caused by the scoring mechanism used by the two models. GC1 uses `scoring='recall'`, whereas GC6 uses `scoring='precision'`, as shown in the following code snippet.

```
# Model GC1
param_grid = {"max_depth": [3,5, 7, 10,None], "n_estimators":[3,5,10,25,50,150], "max_features": [4,7,15,20]}
GC1 = RandomForestClassifier(random_state=2)
grid_search = GridSearchCV(GC1, param_grid=param_grid, cv=5, scoring='recall', verbose=4)
# Model GC6
params = {'n_estimators':[25,50,100,150,200,500],'max_depth':[0.5,1,5,10],'random_state':[1,10,20,42], 'n_jobs':[1,2]}
GC6 = RandomForestClassifier()
grid_search_cv = GridSearchCV(GC6, params, scoring='precision')
```

Further investigation shows, in German Credit dataset, the data rows are personal information about individuals and task is to predict their credit risk. The data items are not balanced when *sex* of the individuals is concerned. The dataset contains 69% data instances of *male* and 31% *female* individuals. When the model is optimized towards recall (GC1) rather than precision (GC6), the total number of true-positives decreases and false-negative increases. Since the number of instances for privileged group (*male*) is more than the unprivileged group (*female*), decrease in the total number of true-positives also increases the probability of unprivileged group to be classified as favorable. Therefore, the fairness of GC1 is more than GC2, although the accuracy is less. Unlike other group fairness metrics, error rate difference (ERD) accounts for false-positive and false-negative rate difference between privileged and unprivileged group. As described before, optimizing the model for recall increases the total number of false-negatives. We found that the percentage of *male* categorized as favorable is less than the percentage of *female* categorized as favorable. Therefore, an increase in the overall false-negative also increased the error rate of unprivileged group, which in turn caused GC1 to be more biased than GC2 in terms of ERD.

From the above discussion, we observed that the model optimization hyperparameter only considers the overall rates of the performance. However, if we split the data instances based on protected attribute groups, then we see the change of rates vary for different groups, which induces bias. The libraries for model construction also do not provide any option to specify model optimization goals specific to protected attributes and make fairer prediction.

Here, we have seen that GC1 has less bias than GC6 by compromising little accuracy. Do all the models achieve fairness by compromising with performance? We found that models can achieve fairness along with high performance. To compare model performance with the amount of bias, we plotted the accuracy and F1 score of the models with the cumulative bias in Figure 3.4. We can see that GC6 is the most efficient model in terms of performance and has less bias than 5 out of 7 other models in German Credit data. AC6 has more accuracy and F1 score than any other models in Adult Census, and exhibits less bias than AC1, AC2, AC4, AC5, and AC7. Therefore, models can have better performance and fairness at the same time.

Finding 2: Libraries for model creation do not explicitly mention fairness concerns in model constructs.

From Figure 3.3, we can see that HC1 and HC2 show difference in most of the fairness metrics, while operating on the same dataset i.e., Home Credit. HC2 is fairer than HC1 with respect to all the metrics except DI. From Table 3.2, we can see that HC1 has positive bias, whereas HC2 exhibit negative bias. This indicates

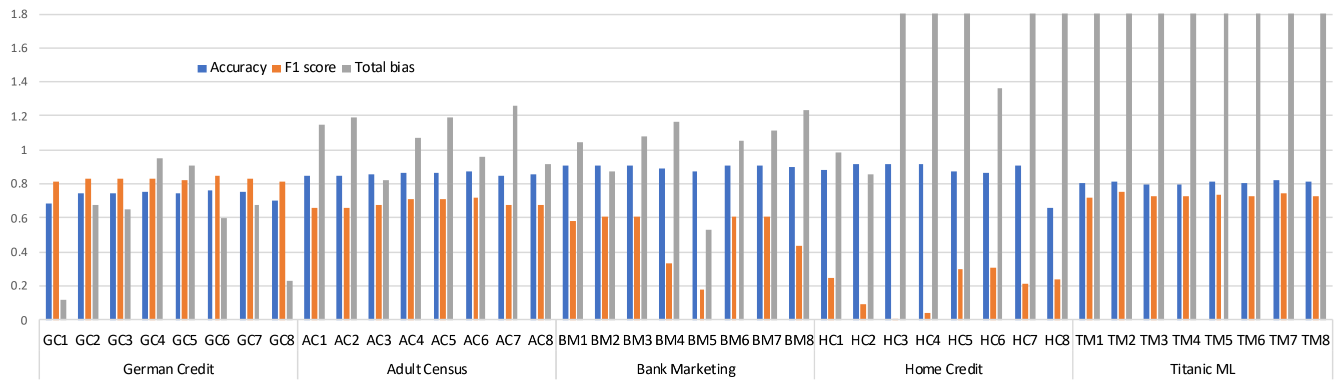

Figure 3.4: Cumulative bias and performance of the machine learning models

that HC1 is biased towards unprivileged group and HC2 is biased towards privileged group. We found that HC1 and HC2 both are using Light Gradient Boost (LGB) model for prediction. The code for building the two models are:

```
# Model HC1
HC1 = lgb.LGBMClassifier(n_estimators=10000, objective='binary', class_weight='balanced', learning_rate=0.05, reg_alpha
    =0.1, reg_lambda=0.1, subsample=0.8, n_jobs=-1, random_state=50)
HC1.fit(X_train, y_train, eval_metric = 'auc', categorical_feature = cat_indices, verbose = 200)
# Model HC2
HC2 = LGBMClassifier(n_estimators=4000, learning_rate=0.03, num_leaves=30, colsample_bytree=.8, subsample=.9, max_depth=7,
     reg_alpha=.1, reg_lambda=.1, min_split_gain=.01, min_child_weight=2, silent=-1, verbose=-1)
HC2.fit(X_train, y_train, eval_metric= 'auc', verbose= 100)
```

We executed both the models with varied hyperparameter combinations and found that `class_weight='balanced'` is causing HC1 not to be biased towards privileged group. By specifying `class_weight`, we can set more weight to the data items belonging to an infrequent class. Higher class weight implies that the data items are getting more emphasis in prediction. When the class weight is set to `balanced`, the model automatically accounts for class imbalance and adjust the weight of data items inversely proportional to the frequency of the class [98, 167]. In this case, HC1 mitigates the *male-female* imbalance in its prediction. Therefore, it does not exhibit bias towards the privileged group (*male*). On the other hand, HC2 has less bias but it is biased towards privileged group. Although we want models to be fair with respect to all groups and individuals, trade-off might be needed and in some cases, bias toward unprivileged may be a desirable trait.

We observed that `class_weight` hyperparameter in LGBMClassifier allows developers to control group fairness directly. However, the library documentation of LGB classifier suggests that this parameter is used for improving performance of the models [167, 182]. Though the library documentation mentions about probability calibration of classes to boost the prediction performance using this parameter, however, there is no suggestion regarding the effect on the bias introduced due to the wrong choice of this parameter.

From the discussions, we can conclude that library developers still do not provide explicit ways to control fairness of the models. Although some parameters directly control the fairness of the models, libraries do not explicitly mention that.

Finding 3: Standardizing features before training models can help to remove disparity between groups in the protected class.

From Figure 3.3 and Figure 3.4, we observe that except BM5, other models in Bank Marketing exhibit similar unfairness. BM5 is a Support Vector Classifier (SVC) tuned using a grid search over given range of parameters. In the modeling pipeline, before training the best found SVC, the features are transformed using `StandardScalar`. Below is the model construction code for BM5 with the best found hyperparameters:

```
tuned_parameters = [{'kernel': ['rbf'], 'gamma': [0.1], 'C': [1]}]
SVC = GridSearchCV(SVC(), tuned_parameters, cv=5, scoring='precision')
# Best found SVC after grid search
# SVC(C=1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma
    =0.1, kernel='rbf', max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001)
model = make_pipeline(StandardScaler(), SVC)
mdl = model.fit(X_train, y_train)
```

We found that the usage of `StandardScalar` in the model pipeline is causing the model BM5 to be fairer. Especially DI of BM5 is 0.14 whereas, the mean of other seven BM models is very high (0.74). `StandardScalar` transforms the data features independently so that the mean value becomes 0 and the standard deviation becomes 1. Essentially, if a feature has variance in orders of magnitude than another feature, the model might learn from the dominating feature more, which might not be desirable [170]. In this case, Bank Marketing dataset has 55 features among which 41 has mean close to 0 ([0, 0.35]). However, *age* is the protected attribute having a mean value 0.97 (*older*: 1, *younger*: 0), since the number of older is significantly more than younger. Therefore, *age* is the dominating feature in these BM models. BM5 mitigates that effect by using standard scaling to all features. Therefore, balancing the importance of protected feature with other features can help to reduce bias in the models. This example also shows the

Table 3.2: Unfairness measures of the models before and after the mitigations

	Model	Before mitigation									After mitigation									
		Acc	F1	DI	SPD	EOD	AOD	ERD	CNT	TI	Acc	F1	DI	SPD	EOD	AOD	ERD	CNT	TI	Rank
German Credit (Sex)*	GC1-RFT	.687	.814	.002	.002	0	.004	.052	-.002	.058	.683	.811	.002	.002	0	.004	-.032	-.002	.058	RAOD/PCE
	GC2-XGB	.743	.828	-.076	-.058	-.039	-.036	.047	-.282	.142	.709	.829	0	0	0	0	.067	0	.057	AORD/PCE
	GC3-XGB	.742	.827	-.105	-.079	-.043	-.065	.036	-.173	.149	.729	.831	-.045	-.040	-.006	-.043	.037	-.095	.100	AR/DPOCE
	GC4-SVC	.753	.832	-.138	-.104	-.081	-.068	.070	-.338	.153	.716	.834	0	0	0	0	.090	0	.057	AORD/PEC
	GC5-EVC	.743	.826	-.148	-.116	-.075	-.089	.067	-.286	.127	.687	.814	0	0	0	0	.112	0	.058	AORD/PEC
	GC6-RFT	.761	.845	-.103	-.083	-.023	-.085	.005	-.183	.121	.759	.844	-.071	-.058	-.023	-.085	-.027	-.183	.121	RD/APCEO
	GC7-XGB	.751	.831	-.073	-.056	.009	-.072	-.033	-.293	.144	.709	.829	0	0	0	0	.047	0	.057	ADR/POCE
	GC8-KNN	.698	.815	.003	.002	0	.011	.081	-.041	.090	.702	.825	0	0	0	0	.086	0	.057	AR/DPCOE
Adult Census (Race)*	AC1-LRG	.845	.657	-.654	-.104	-.100	-.069	-.050	-.045	.127	.261	.399	.023	.023	.017	.021	.120	-.019	.040	ORCDAP/E
	AC2-RFT	.846	.657	-.582	-.098	-.047	-.046	-.060	-.236	.119	.787	.249	-.354	-.014	.007	.003	-.086	-.005	.232	AROC/DPE
	AC3-GBC	.858	.677	-.496	-.079	-.041	-.031	-.045	-.010	.120	.858	.675	-.131	-.024	-.041	-.031	-.004	-.010	.120	ROAC/DPE
	AC4-CBC	.869	.712	-.616	-.102	-.077	-.056	-.044	-.069	.107	.805	.683	-.127	-.044	.080	.044	-.001	-.102	.082	ORAC/PDE
	AC5-XGB	.867	.708	-.588	-.097	-.073	-.051	-.043	-.224	.111	.865	.705	-.203	-.039	-.073	-.051	-.002	-.224	.111	ROAC/PDE
	AC6-XGB	.871	.717	-.570	-.096	-.044	-.036	-.047	-.062	.106	.808	.691	-.132	-.046	.072	.044	.009	-.094	.078	ORAC/PDE
	AC7-RFT	.852	.678	-.615	-.104	-.078	-.059	-.051	-.235	.117	.638	.329	-.289	-.024	-.005	-.009	-.039	-.009	.187	AORCD/PE
	AC8-DCT	.853	.675	-.519	-.086	-.040	-.035	-.050	-.068	.121	.852	.673	-.153	-.029	-.040	-.035	-.010	-.068	.121	ROAC/DPE
Bank Marketing (Age)	BM1-XGB	.906	.582	.627	.087	.074	.053	.051	-.078	.074	.905	.581	.274	.032	.074	.053	.017	-.078	.074	ROCPD/EA
	BM2-LGB	.908	.606	.593	.083	.004	.022	.069	-.034	.072	.772	.498	.076	.026	-.037	-.037	-.031	-.040	.066	ORDC/PAE
	BM3-GBC	.908	.604	.688	.100	.083	.056	.051	-.032	.072	.852	.529	.066	.013	-.059	-.052	006	-.089	078	CODR/APE
	BM4-XGB	.887	.330	.810	.048	.067	.042	.074	-.010	.111	.887	.328	.442	.022	.067	.042	.001	-.010	.111	RCA/OPDE
	BM5-SVC	.875	.175	.139	.003	-.077	-.031	.126	-.032	.126	.873	.002	.139	0	-.001	0	.110	0	.136	ERCDO/AP
	BM6-GBC	.908	.612	.698	.105	.030	.038	.076	-.033	.071	.795	.521	.110	.034	-.072	-.053	-.019	-.039	.065	OCRD/PAE
	BM7-XGB	.910	.611	.713	.107	.051	.052	.072	-.047	.070	.829	.485	.022	.004	-.037	-.044	-.007	-.122	.085	CODRA/PE
	BM8-RFT	.899	.435	.834	.066	.091	.058	.064	-.023	.097	.795	.462	.289	.042	-.048	-.027	.005	-.052	.073	ORACDP/E

Reweighing (R) DI Remover (D) Adversarial Debiasing (A) Prejudice Remover (P) Equalized Odds(E)
Calibrated Equalized Odds (C) Reject Option Classification (O)

Table 3.2 Continued

	Model	Before mitigation									After mitigation									
		Acc	F1	DI	SPD	EOD	AOD	ERD	CNT	TI	Acc	F1	DI	SPD	EOD	AOD	ERD	CNT	TI	Rank
Home Credit (Sex)	HC1-LGB	.883	.249	.574	.046	.065	.052	.051	-.110	.083	.238	.132	-.025	-.002	-.003	-.002	-.020	-.006	.030	APECR/OD
	HC2-LGB	.920	.094	-.698	-.006	-.016	-.010	-.032	-.012	.081	.919	.002	.076	0	0	0	-.033	0	.084	PECROA/D
	HC3-GNB	.913	.010	.974	.999	.007	.005	.006	-2.449	0	.732	.194	.181	.857	.047	.019	.031	-2.285	0	OA/DECPR
	HC4-XGB	.919	.046	.868	.994	.003	.013	.007	-2.482	0	.918	.012	-.103	.998	0	-.003	-.002	-2.468	0	CEDRP/OA
	HC5-CBC	.870	.302	.744	.865	.085	.140	.106	-2.524	0	.552	.075	-.134	.999	-.025	-.017	-.021	-2.772	.001	ACEPR/DO
	HC6-CBC	.869	.305	.735	.085	.144	.107	.068	-.147	.080	.583	.074	.021	0	0	0	.007	0	.056	ACPER/DO
	HC7-XGB	.911	.211	.953	.953	.033	.084	.054	-2.533	0	.907	.090	.408	.966	.009	-.052	-.019	-2.453	0	ECPR/DOA
	HC8-RFT	.661	.239	.383	.719	.147	.129	.133	-2.449	.001	.645	.226	.337	.681	.133	.098	.112	-2.426	.001	CPRD/AEO
Titanic ML (Sex)	TM1-XGB	.807	.720	-2.247	-.705	-.631	-.559	-.056	-.341	.153	.649	.580	-.082	-.039	.027	.177	.115	-.272	.189	OAERDP/C
	TM2-RFT	.816	.753	-2.013	-.709	-.635	-.515	.022	-.293	.142	.644	.566	-.106	-.045	.059	.166	.023	-.269	.223	OAERDP/C
	TM3-EBG	.799	.725	-2.125	-.674	-.637	-.514	-.017	-.333	.165	.647	.572	-.108	-.045	.031	.148	.050	-.317	.223	OAERD/PC
	TM4-LRG	.800	.732	-2.439	-.808	-.729	-.694	-.051	-.381	.144	.658	.577	-.075	-.034	.072	.160	.038	-.327	.207	OAEPRD/C
	TM5-GBC	.816	.740	-2.268	-.708	-.647	-.542	-.022	-.357	.151	.651	.572	-.087	-.033	.097	.174	.029	-.332	.205	OAERD/CP
	TM6-XGB	.804	.730	-1.948	-.665	-.583	-.499	-.042	-.345	.146	.625	.568	-.079	-.038	.075	.157	.092	-.367	.190	OAERD/CP
	TM7-RFT	.825	.747	-2.232	-.639	-.555	-.411	-.029	-.285	.161	.653	.577	-.099	-.043	.100	.188	.003	-.261	.219	OAERDP/C
	TM8-RFT	.814	.732	-2.306	-.716	-.633	-.563	-.051	-.321	.149	.649	.596	-.082	-.042	.011	.166	.157	-.327	.172	OAERD/PC

*Experiment has been conducted for multiple protected attributes. RFT: Random Forest, XGB: XGBoost, SVC: Support Vector Classifier, EVC: Ensemble Voting Classifier, KNN: K-Nearest Neighbors, LRG: Logistic Regression, GBC: Gradient Boosting Classifier, CBC: Cat Boost Classifier, DCT: Decision Tree, LGB: Light Gradient Boost, GNB: Gaussian Naive Bayes, EBG: Ensemble Bagging. Mitigation techniques applied to the models are as follows. Result is shown for the best mitigation. Rank of mitigation uses acronym below (mitigations before '/' have been able to mitigate bias, rest have not.)

Reweighing (R) DI Remover (D) Adversarial Debiasing (A) Prejudice Remover (P) Equalized Odds(E)
Calibrated Equalized Odds (C) Reject Option Classification (O)

importance of understanding the underlying properties of protected features and their effectiveness on prediction.

Finding 4: Dropping a feature from the dataset can change the model fairness effectively.

Both the models AC5 and AC6 are using XGB classifier for prediction but AC6 is fairer than AC5. Among the metrics, in terms of consistency (CNT), AC5 shows bias 3.61 times more than AC6. We investigated the model construction and found that AC5 and AC6 differ in three constructs: features used in the model, number of trees used in the random forest, and learning rate of the classifier. We observed that the number of trees and learning rate did not change the bias of the models. In AC5, the model excluded one feature from the training data. Bank Marketing dataset contains personal information about individuals and predicts whether the person has an annual income more than 50K dollars or not. In AC5, the model developer dropped one feature that contains number of years of education, since there is other categorical feature which represents education of the person (e.g., bachelors, doctorate, etc.). AC6 is using all the features in the dataset. CNT measures the individual fairness of the models i.e., how two similar individuals (not necessarily from different groups of protected attribute class) are classified to different outcomes. Therefore, dropping the number of years of education is causing the model to classify similar individuals to different outcome, which in turn generating individual unfairness.

Finding 5: Different metrics are needed to understand bias in different models.

From Figure 3.3, we can see that the models show different patterns of bias in terms of different fairness metrics. For example, compared to any Bank Marketing models, BM5 has disparity impact (DI) less than half but the error rate difference (ERD) more than twice. If the model developer only accounts for DI, then the model would appear fairer than what it actually is. As another example, GC6 is fairer than 90% of all the models in terms of total bias but if we only consider consistency (CNT), GC6 is fairer than only 50% of all the models. However, previous studies show that achieving fairness with respect to all the metrics is difficult and for some pair of metrics, mathematically impossible [124, 48, 20]. Also, the definition of fairness can vary depending on the application context and the stakeholders. Therefore, it is important to report on

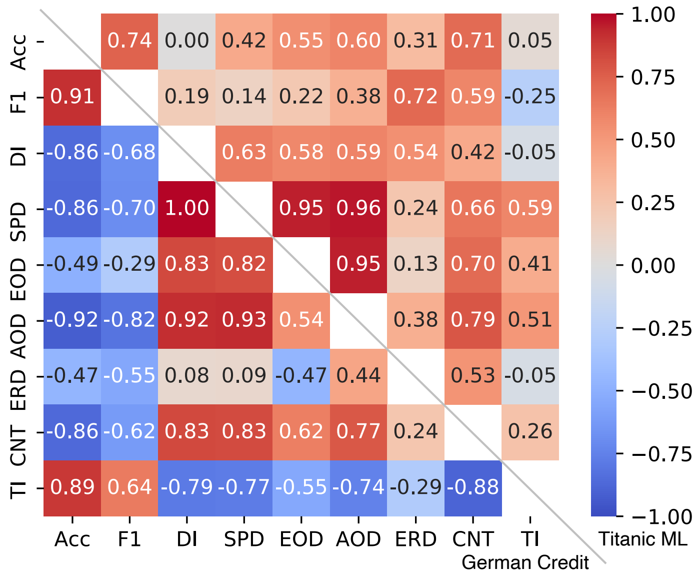

Figure 3.5: Corelation between the fairness metrics. Bottom diagonal is for German Credit models, top diagonal is for Titanic ML models.

comprehensive set of fairness measures and evaluate the trade-off between the metrics to build fairer. We plotted the correlation between different metrics from two datasets in Figure 3.5. A few metric pairs have a similar correlation in both the datasets such as (SPD, EOD), (SPD, AOD). This is understandable from the definitions of these metrics because they are calculated using same or correlated group conditioned rates (true-positives and false-positives). Although there are many metric pairs which are positively or negatively correlated, there is no pattern in correlation values between the two datasets. For instance, CNT and TI are highly negatively correlated in German Credit models but positively correlated in Titanic ML models. Therefore, we need a comprehensive set of metrics to evaluate fairness.

Finding 6: Except DI, EOD, and AOD, all the fairness measures remain consistent over multiple training and prediction.

To measure the stability of the fairness and performance metrics, we computed the standard deviation of each metric over 10 runs similar to [65]. In each run, the dataset is shuffled before the train-test split, and model is trained on a new randomized training set. We have seen that the models are stable for the performance metrics and most of the fairness metrics. In particular, the average of the standard deviations of accuracy, F1

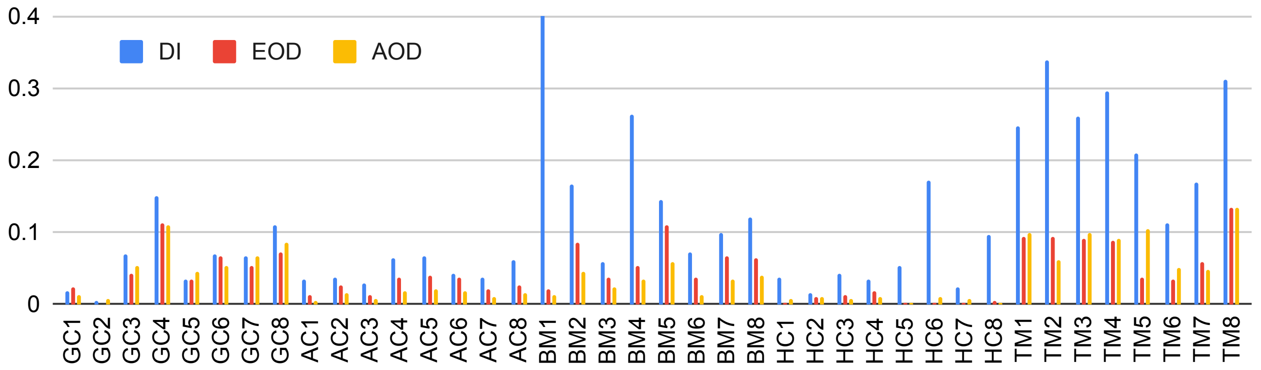

Figure 3.6: Standard deviation of the fairness metrics: DI, EOD and AOD over multiple experiments. Other metrics have very low standard deviation.

score, DI, SPD, EOD, AOD, ERD, CNT and TI over all the models are 0.01, 0.01, 0.12, 0.03, 0.04, 0.04, 0.03, 0.01, 0.01, respectively. Except for DI, EOD and AOD, the average standard deviation is very low (less than 0.03). For these three metrics, we plotted the standard deviations in Figure 3.6. We can see that the trend of standard deviations is similar to the models of a specific dataset. In our benchmark, the largest dataset is Home Credit, which has the lowest standard deviation and the smallest dataset is Titanic ML, which has the most. Since in larger dataset, even after shuffling the training data remains more consistent, the deviation is less. On the other hand, the Titanic ML dataset is the smallest in size, having 891 data instances. The class distribution of data instances do not remain consistent when a random training set is chosen. Therefore, while dealing with smaller datasets, it is important to choose a training set that represents the original data and evaluate fairness multiple times.

DI has more standard deviation than other metrics. DI is computed using the ratio of two probabilities, P_u/P_p, where P_u is the probability of unprivileged group getting favorable label, and P_p is the probability of privileged group getting favorable label. Even the probability difference is very low, the value of DI can be very high. Therefore, DI fluctuates more frequently than other metrics.

Finding 7: A fair model with respect to one protected attribute is not necessarily fair with respect to another protected attribute.

To understand the behavior of the same models on different protected attributes, we analyzed the fairness of German Credit and Adult Census models on two protected attributes. In Figure 3.7, we plotted the fairness

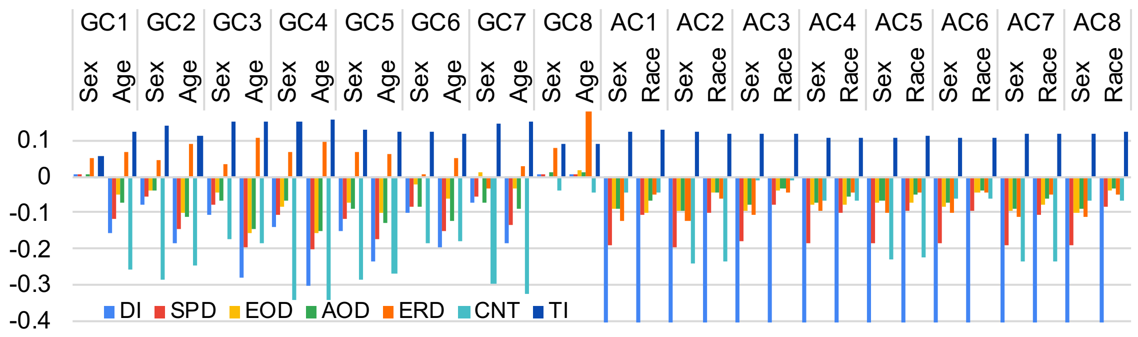

Figure 3.7: Fairness of ML models with respect to different protected attributes

measures of German Credit models on *sex* and *age* and Adult Census models on *sex* and *race*. We found that the models can show different fairness when different protected attribute is considered. The total bias exhibited by German Credit dataset are: for *sex* attribute 4.82 and for *age* attribute 7.72. For Adult Census, the total bias are: for *sex* attribute 15.15 and for *race* attribute 8.56. However, most of the models exhibit similar trend of difference in the fairness when considering two different attributes.

GC1 and GC6 show cumulative bias 0.12 and 0.60 when *sex* is considered. Surprisingly, GC1 and GC6 shows cumulative bias 0.85 and 0.88 when *age* is considered. GC1 is much fairer model than GC6 in the first case but in the second case, the fairness is almost similar. We discussed the behavior of these two models in Finding 1 and explained how GC1 is fairer when *sex* is the protected attribute. However, the fair prediction does not persist for the *age* because there is no imbalance in German Credit with respect to *age* groups. Therefore, GC1 and GC6 show similar fairness when *age* is considered.

3.5 Bias Mitigation

In this section, we investigated the fairness results of the models after applying bias mitigation techniques. We employed 7 different bias mitigation algorithms separately on 40 models and compared the fairness results with the original fairness exhibited by the models. For each model, we selected the most successful mitigation algorithm and plotted the fairness values after mitigation in Figure 3.8. We observed that similar to Figure 3.3, the fairness patterns are similar for the models in a dataset. DI, SPD, and CNT are the most difficult metrics to mitigate.

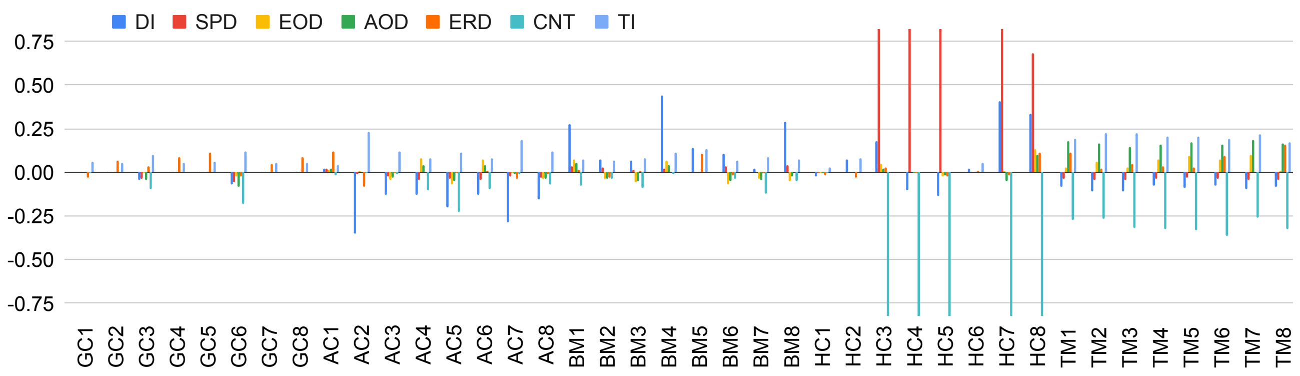

Figure 3.8: The fairness exhibited by the models after applying the most effective bias mitigation techniques

To understand the root causes of unfairness, we focused on the models which exhibit more or less bias and then investigated the effects of different mitigation algorithms. Here, among the mitigation algorithms, the preprocessing techniques operate on the training data and retrain the original model to remove bias. On the other hand, post-processing techniques do not change the training data or original model but change the prediction made by the model. The in-processing techniques do not alter the dataset or prediction result but employ completely new modeling technique.

Finding 8: Models with effective preprocessing mitigation technique is preferable than others.

We found that Reweighing algorithm has effectively debiased many models: GC1, GC6, AC3, AC5, AC8, BM1 and BM4. These models produce fairer results when the dataset is pre-processed using Reweighing. In other words, these models do not propagate bias themselves. In other cases where pre-processing techniques are not effective, we had to change the model or alter the prediction, which implies that bias is induced or propagated by the models. Another advantage is that in these models, after mitigations the models have retained the accuracy and F1 score. Other mitigation techniques often hampered the performance of the model. For a few other models (GC3, GC8, AC1, AC2, AC4, AC6, BM2, BM5, BM8), Reweighing has been the second most successful mitigation algorithm. Among these models, in AC1, AC2, BM2, and BM5, the most successful algorithm to mitigate bias loss accuracy or F1 score at least 22%. In all of these cases, Reweighing has retained both accuracy and F1 score.

Finding 9: Models with more bias are debiased effectively by post-processing techniques, whereas originally fairer models are debiased effectively by preprocessing or in-processing techniques.

From Table 3.2, we can see that 21 out of 40 models are debiased by one of the three post-processing algorithms i.e., Equalized odds (EO), Calibrated equalized odds (CEO), and Reject option classifier (ROC). These algorithms have been able to mitigate bias (not necessarily the most successful) in 90% of the models. Especially, ROC and CEO are the dominant post-processing techniques. ROC takes the model prediction, and gives the favorable outcome to the unprivileged group and unfavorable outcome to privileged group with a certain confidence around the decision boundary [114]. CEO takes the probability distribution score generated by the classifier and find the probability of changing outcome label and maximize equalized odds [148]. EO also changes the outcome label with certain probability obtained by solving a linear program [84]. We found that these post-processing methods have been able to mitigate bias more effectively when the original model produces more biased results. From Figure 3.4, we can see that the most biased 5 models are TM4, TM8, TM5, TM1, HC7, where the post-processing has been the most successful algorithms. On the contrary, in case of the 5 least biased model (GC1, GC8, BM5, GC6, GC3), rather than mitigating, all three post-processing techniques increased bias.

In Table 3.2, we have shown the rank of mitigation algorithms to debias each model. In Table 3.3, we have shown the mean of the ranks of each mitigation algorithms, where rank of most successful algorithm is 1 and least is 7. We can see that for most biased models, Reject option classification and Equalized odds have been more successful than all others. For the least biased models, both preprocessing algorithms and Adversarial Debiasing have been effective, and the post-processing algorithms have been ineffective.

3.6 Impact of Mitigation

While mitigating bias, there is a chance that the performance of the model is diminished. The most successful algorithm in debiasing a model does not always give good performance. So, often the developers have to trade-off between fairness and performance. In this section, we investigated the answer to RQ3. What

Table 3.3: Mean rank of each bias mitigation algorithm for 10 least biased models (LBM), 10 most biased models (MBM), and overall

Stage	Algorithms	LBM	MBM	All
Preprocessing	Reweighing (R)	2.1	4.5	3.03
	Disparate Impact Remover (D)	3.7	4.8	4.58
In-processing	Adversarial Debiasing (A)	3	2.9	3
	Prejudice Remover Regularizer (P)	4.5	5.3	4.98
Post-processing	Equalized Odds (E)	5.8	2.8	5.18
	Calibrated Equalized Odds (C)	4.8	5.1	4.33
	Reject Option Classification (O)	4.1	2.6	2.93

are the impacts when the bias mitigation algorithms are applied to the models? We have analyzed the accuracy and F1 score of the models after applying the mitigation algorithms.

First, for each model, we analyzed the impacts of the most effective mitigation algorithms in removing bias. In Figure 3.9, we plotted the change in accuracy, F1 score, and total bias when the most successful mitigating algorithms are applied. We can see that while mitigating bias, many models are losing their performance. From Table 3.2, pre-processing algorithms, especially Reweighing has been the most effective in model GC1, GC6, AC3, AC5, AC8, BM1, and BM3. From Figure 3.9, these models always retain their performance after mitigation.

Finding 10: When mitigating bias effectively, in-processing mitigation algorithms show uncertain behavior in their performance.

Among in-processing algorithms, Adversarial debiasing has been the most effective in 11 models (GC2, GC3, GC4, GC5, AC2, AC7, HC1, HC5, HC6), and Prejudice remover has been the most effective in 1 model (HC2). We found that for German Credit models Adversarial debiasing has been effective without losing performance. But in other cases, AC1, AC7, HC1, and HC7, the accuracy has decreased at least 21.4%. In HC2, Prejudice remover also loses F1 score while mitigating the bias. Since, in-processing techniques employ new model and ignore the prediction of the original model, in all situations (dataset and task), it is not giving better performance. In our evaluation, adversarial debiasing is giving good performance with German Credit dataset but not on Adult Census or Home Credit dataset. Therefore, in-processing techniques

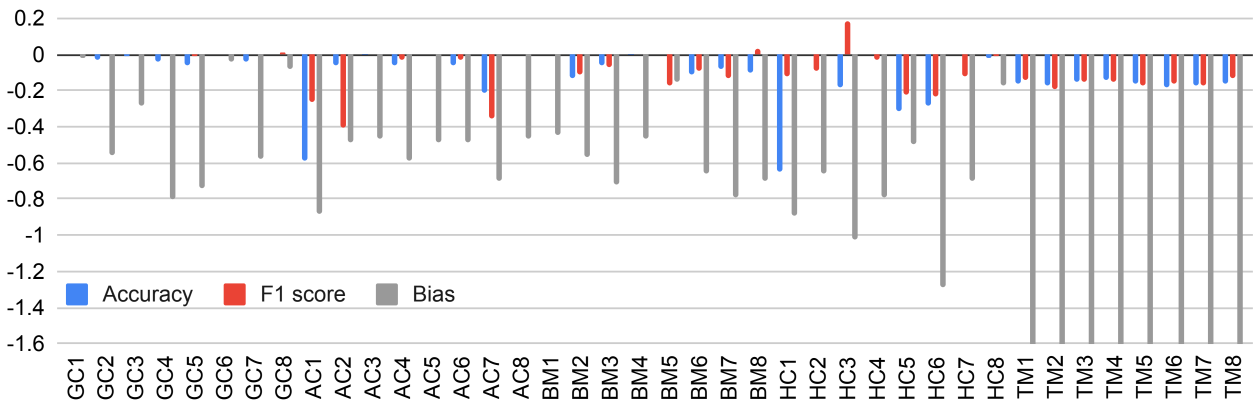

Figure 3.9: Change of performance and bias after applying bias mitigation techniques (negative value indicates reduction)

are not appropriate when we can not change the original modeling. Also, since these techniques are uncertain in retaining performance, the developers should be careful about the accuracy of prediction after the intervention.

Finding 11: Although post-processing algorithms are the most dominating in debiasing, they are always diminishing the model accuracy and F1 score.

From Table 3.2, we can see that in 21 out of 40 models, one of the post-processing algorithms are being the most successful. But in all of the cases they are losing performance. The average accuracy reduction in these models is 7.49% and average F1 decrease is 10.07%. For example, in AC1, the most bias mitigating algorithm is Reject option classification but the model is loosing 26.1% accuracy and 40% F1 score. In these cases, developers should move to the next best mitigation algorithm. In a few other cases such as HC8, the Reject Option classification mitigates bias with only 1.6% loss in accuracy and 1.3% loss in f1 score. In such situations, post-processing techniques can be applied to mitigate the bias.

Finding 12: Trade-off between performance and fairness exists, and post-processing algorithms have most competitive replacement.

Since some most mitigating algorithms are having performance reduction, for each model, we compared the most successful algorithm with the next best mitigation algorithm in Figure 3.10. We found that for 18 out of

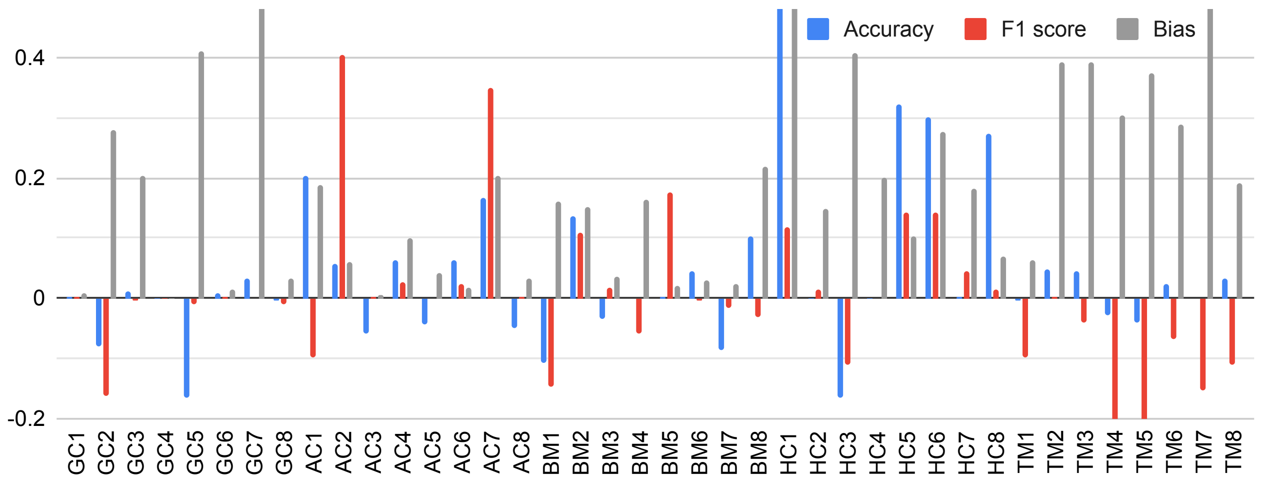

Figure 3.10: Change of performance and bias between the 1st and 2nd most successful mitigation algorithms (negative value indicates reduction)

40 models, the performance of the 2nd ranked algorithm is same or better than the 1st ranked algorithm. Among them, in AC4, AC6, BM5, HC5, and HC8, the 2nd ranked algorithm has bias very close (not more than 0.1) to the 1st ranked one. All of these, except HC5, the 1st ranked bias mitigation algorithm is a post-processing technique. We observe that competitive alternative mitigation technique is more common for post-processing mitigation algorithms. Therefore, if we increase the tolerable range of bias, then other mitigation techniques would be better alternative in terms of performance. In addition, in some domains, bias towards one group might be preferred over other group. In those cases, for choosing mitigation techniques, developers should consider the direction of the metrics rather than considering absolute value.

3.7 Discussion

3.7.1 Threats to Validity

Benchmark Creation. To avoid experimenting on low-quality kernels, we have only considered the kernels with more than 5 votes. In addition, we have excluded the kernels where the model accuracy is very low (less than 65%). Finally, we have selected the top voted ones from the list. We have also verified that the collected kernels are runnable. To ensure the models collected from Kaggle are appropriate for fairness study, we have first selected the fairness analysis datasets from previous works and searched models for those datasets. Finally, we have searched competitions that use dataset with protected attributes used in the literature.

Fairness and performance evaluation. Our collected models give the same performance, as mentioned in the corresponding Kaggle kernels. For evaluating fairness and applying mitigation algorithms we have used AIF 360 toolkit [19] developed by IBM. Bellamy *et al.* presented fairness results (4 metrics) for two models (Logistic regression and Random forest) on Adult Census dataset with protected attribute *race* [19]. We have done experiment with the same setup and validated our result [19]. Similar to [65], for each metric, we have evaluated 10 times and taken the mean of the values. The stability comparison of the results is shown in Section 3.4.

Fairness comparison. As different metrics are computed based on different definitions of fairness, we have compared bias using a specific metric or cumulatively. Finally, in this paper, we have focused on comparing fairness of different models. Therefore, for each dataset, we followed the same method to pre-process training and testing data.

3.8 Summary

ML fairness has received much attention recently. However, ML libraries do not provide enough support to address the issue in practice. In this paper, we have empirically evaluated the fairness of ML models and discussed our findings of software engineering aspects. First, we have created a benchmark of 40 ML models from 5 different problem domains. Then, we have used a comprehensive set of fairness metrics to measure fairness. After that, we have applied 7 mitigation techniques on the models and computed the fairness metric again. We have also evaluated the performance impact of the models after mitigation techniques are applied. We have found what kind of bias is more common and how they could be addressed. Our study also suggests that further SE research and library enhancements are needed to make fairness concerns more accessible to developers.

CHAPTER 4. THE ART AND PRACTICE OF DATA SCIENCE PIPELINES

Increasingly larger number of software systems today are including data science components for descriptive, predictive, and prescriptive analytics. The collection of data science stages from acquisition, to cleaning/curation, to modeling, and so on are referred to as *data science pipelines*. Most of the research in understanding fairness in classification tasks has considered the whole pipeline as a monolithic components. However, there could be many stages with several components and operators in the pipeline. To facilitate research and practice on data science pipelines, it is essential to understand their nature. What are the typical stages of a data science pipeline? How are they connected? Do the pipelines differ in the theoretical representations and that in the practice? Today we do not fully understand these architectural characteristics of data science pipelines. In this work, we present a three-pronged comprehensive study to answer this for the state-of-the-art, data science in-the-small, and data science in-the-large. Our study analyzes three datasets: a collection of 71 proposals for data science pipelines and related concepts in *theory*, a collection of over 105 implementations of curated data science pipelines from Kaggle competitions to understand data science *in-the-small*, and a collection of 21 mature data science projects from GitHub to understand data science *in-the-large*. Our study has led to three representations of data science pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large.

4.1 Introduction

Data science processes, also called *data science stages* as in stages of a pipeline, for descriptive, predictive, and prescriptive analytics are becoming integral components of many software systems today. The data science stages are organized into a *data science pipeline*, where data might flow from one stage in the pipeline to the next. These data science stages generally perform different tasks such as data acquisition, data preparation, storage, feature engineering, modeling, training, evaluation of the machine learning model, etc. In order to design and build software systems with data science stages effectively, we must understand the structure of the data science pipelines. Previous work has shown that understanding the structure and patterns used in existing systems and literature can help build better systems [175, 67]. In this work, we have taken the first step to understand the structure and patterns of DS pipelines.

Fortunately, we have a number of instances in both the state-of-the-art and practice to draw observations. In the literature, there have been a number of proposals to organize data science pipelines. We call such proposals *DS Pipelines in theory*. Another source of information is Kaggle, a widely known platform for data scientists to host and participate in DS competitions, share datasets, machine learning models, and code. Kaggle contains a large number of data science pipelines, but these pipelines are typically developed by a single data scientist as small standalone programs. We call such instances *DS Pipelines in-the-small*. The third source of DS pipelines are mature data science projects on GitHub developed by teams, suitable for reuse. We call such instances *DS Pipelines in-the-large*.

This work presents a study of DS pipelines in theory, in-the-small, and in-the-large. We studied 71 different proposals for DS pipelines and related concepts from the literature. We also studied 105 instances of DS pipelines from Kaggle. Finally, we studied 21 matured open-source data science projects from GitHub. For both Kaggle and GitHub, we selected projects that make use of Python to ease comparative analysis. In each setting, we answer the following overarching questions.

1. **Representative pipeline:** What are the stages in DS pipeline and how frequently they appear?
2. **Organization:** How are the pipeline stages organized?
3. **Characteristics:** What are the characteristics of the pipelines in a setting and how does that compare with the others?

This work attempts to inform the terminology and practice for designing DS pipeline. We found that DS pipelines differ significantly in terms of detailed structures and patterns among theory, in-the-small, and in-the-large. Specifically, a number of stages are absent in-the-small, and the pipelines have a more linear structure with an emphasis on data exploration. Out of the eleven stages seen in theory, only six stages are present in pipeline in-the-small, namely *data collection*, *data preparation*, *modeling*, *training*, *evaluation*, and *prediction*. In addition, pipelines in-the-small do not have clear separation between stages which makes the maintenance harder. On the other hand, the DS pipelines in-the-large have a more complex structure with feedback loops and sub-pipelines. We identified different pipeline patterns followed in specific phase (development/post-development) of the large DS projects. The abstraction of stages are stricter in-the-large having both loosely- and tightly-coupled structure.

Our investigation also suggest that DS pipeline is a well used software architecture but often built in ad hoc manner. We demonstrated the importance of standardization and analysis framework for DS pipeline following the traditional software engineering research on software architecture and design patterns [175, 129, 142]. We contributed three representations of DS pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large that would facilitate building new DS systems. We anticipate our results to inform design decisions made by the pipeline architects, practitioners, and software engineering teams. Our results will also help the DS researchers and developers to identify whether the pipeline is missing any important stage or feedback loops (e.g., *storage* and *evaluation* are missed in many pipelines).

The rest of this chapter is organized as follows: in Section 4.2, we present our study of DS pipelines in theory. Section 4.3 describes our study of DS pipelines in-the-small. In Section 4.4, we describe our study of DS pipelines in-the-large.

4.2 DS Pipeline in Theory

Data Science. Data Science (DS) is a broad area that brings together computational understanding, inferential thinking, and the knowledge of the application area. Wing [203] argues that DS studies how to extract value out of data. However, the value of data and extraction process depends on the application and context. DS includes a broad set of traditional disciplines such as data management, data infrastructure building, data-intensive algorithm development, AI (machine learning and deep learning), etc., that covers both the fundamental and practical perspectives from computer science, mathematics, statistics, and domain-specific knowledge [21, 187]. DS also incorporates the business, organization, policy and privacy issues of data and data-related processes. Any DS project involves three main stages: data collection and preparation, analysis and modeling, and finally deployment [202]. DS is also more than statistics or data mining since it incorporates understanding of data and its pattern, developing important questions and answering them, and communicating results [187].

Data Science Pipeline. The sequential DS stages from acquisition, to cleaning/curation, to modeling, and so on are referred to as *data science pipeline*. A DS pipeline may consist of several stages and connections between them. The stages are defined to perform particular tasks and connected to other stage(s) with input-output relations [9]. However, the definitions of the stages are not consistent across studies in the literature. The terminology vary depending on the application context and focus. In addition, there is no universally agreed definition of DS pipeline that includes all the stages [173]. Different study in the literature presented DS pipeline based on their context and desiderata. No study has been conducted to unify the notions DS pipeline and collect the concepts [173]. While designing a new DS pipeline [204], dividing roles in DS teams [121], defining software process in data-intensive setting [197], identifying best practices in AI and modularizing DS components [9], it is important to understand the current state of the DS pipeline, its variations and different stages. To understand the DS pipelines and compare them, we collected the available pipelines from the literature and conducted an empirical study to unify the stages with their subtasks. Then we created a representative DS pipeline with the definitions of the stages. Next, we present the methodology and results of our analysis of DS pipelines in theory.

4.2.1 Methodology

Collecting Data Science Pipelines We searched for the studies published in the literature and popular press that describes DS pipelines. We considered the studies that described both end-to-end DS pipeline or a partial DS pipeline specific to a context. First, we searched for peer-reviewed papers published in the last decade i.e., from 2010 to 2020. We searched the terms "*data science pipeline*", "*machine learning pipeline*", "*big data lifecycle*", "*deep learning workflow*", and the permutation of these keywords in IEEE Xplore, ACM Digital Library and Google Scholar. From a large pool, we selected 1,566 papers that fall broadly in the area of computer science, software engineering and data science. Then we analyzed each article in this pool to select the ones that propose or describe a DS pipeline. We found many papers in this collection use the terms (e.g., ML lifecycle), but do not contain a DS pipeline. We selected the ones that contain DS pipeline and extracted the pipelines (screenshot/description) as evidence from the article. The extracted raw pipelines are available in the artifact accompanied by this paper [23]. Thus, we found 46 DS pipelines that were published in the last decade.

Besides peer-reviewed papers, by searching the keywords on web, we collected the DS pipelines from US patent, industry blogs (e.g., Microsoft, GoogleCloud, IBM blogs), and popular press published between 2010 and 2020. After manual inspection, we found 25 DS pipelines from this grey literature. Thus, we collected 71 subjects (46 from peer reviewed articles and 25 from grey literature) that contain DS pipeline. We used an open-coding method to analyze these DS pipelines in theory [23] .

Labeling Data Science Pipelines In the collected references, DS pipeline is defined with a set of stages (*data acquisition*, *data preparation*, *modeling*, etc.) and connections among them. Each stage in the pipeline is defined for performing a specific task and connected to other stages. However, not all the studies depict DS pipelines with the same set of stages and connections. The studies use different terminologies for defining the stages depending on the context. To be able to compare the pipelines, we had to understand the definitions and transform them into a canonical form. For a given DS pipeline, identifying their stages and mapping them to a canonical form is often challenging. The sub-tasks, overall goal of the project, utilities affect the understanding of the pipeline stages. To counter these challenges, we used an open-coding method to label the stages of the pipelines.

Two authors labeled the collected DS pipelines into different criteria. Each author read the article, understood the pipeline, identified the stages, and labeled them. In each iteration, the raters labeled 10% of the subjects (7-8 pipelines). The first 8 subjects were used for training and forming the initial labels. After each iteration, we calculated the Cohen's Kappa coefficient [195], identified the mismatches, and resolved them in the presence of a moderator, who is another author. Thus, we found the representative DS pipeline after rigorous discussions among the raters and the moderator. The methodology of this open-coding procedure is shown in Figure 4.1. The entire labeling process was divided into two phases: 1) training, and 2) independent labeling.

Training: The two raters were trained on the goal of this project and their roles. We randomly selected eight subjects for training. First, the raters and the moderator had discussions on three subjects and identified

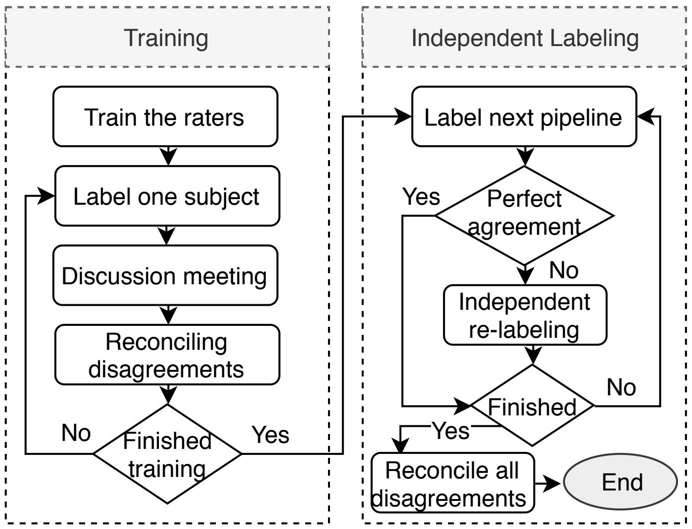

Figure 4.1: Labeling method for DS pipelines in theory

the stages in their DS pipeline. Thus, we formed the commonly occurred stages and their definitions, which were updated through the entire labeling and reconciliation process later. After the initial discussion and training, the raters were given the already created definitions of the stages and one pipeline from the remaining five for training. The raters labeled this pipeline independently. After labeling the pipeline, we calculated the agreement and conducted a discussion session among the raters and the moderator. In this session, we reconciled the disagreements and updated the labels with the definitions. We continued the training session until we got perfect agreement independently. The inter-rater agreement was calculated using Cohen's Kappa coefficient [195]. A higher κ ([0, 1]) indicates a better agreement. The interpretation of of κ is shown in Table 4.1a. In the discussion meetings, the raters discussed each label (both agreed and disagreed ones) with the other rater and moderator, argued for the disagreed ones and reconciled them. In this way, we came up with most of the stages and a representative terminology for each stage including the sub-tasks.

Independent labeling: After completing the training session, the rest of the subjects were labeled independently by the raters. The raters labeled the remaining 63 labels: 7 subjects (10%) in each of the 9 iterations. The distribution of κ after each independent labeling iteration is shown in Table 4.1b. In each iteration, first, the raters had the labeling session, and then the raters and moderator had the reconciliation session.

Range (κ)	**Agreement level**
0.00 - 0.20	Slight agreement
0.21 - 0.40	Fair agreement
0.41 - 0.60	Moderate agreement
0.61 - 0.80	Substantial agreement
0.81 - 1.00	Perfect agreement

(a) Interpretation of Kappa (κ)

Iteration #	κ	**Iteration #**	κ
1	0.67	6	0.91
2	0.74	7	0.87
3	0.82	8	0.90
4	0.84	9	0.94
5	0.84	10	0.91

(b) Agreement in different stages

Table 4.1: Labeling agreement calculation

Labeling. The raters labeled separately so that their labels were private, and they did not discuss while labeling. The raters identified the stages and connections between them, and finally labeled whether the DS pipeline involves processes related to cyber, physical or human component in it. In independent labeling, we found almost perfect agreement (κ = 0.83) on average. Even after high agreement, there were very few disagreements in the labels, which were reconciled after each iteration.

Reconciling. Reconciliation happened for each label for the subject studies in the training session, and the disagreed labels for the studies in independent labeling session. In training session, the reconciliation was done in discussion meetings among the raters and the moderator, whereas for the independent labels, reconciliation was done by the moderator after separate meetings with the two raters. For reconciliation, the raters described their arguments for the mislabeled stages. For a few cases, we had straightforward solution to go for one label. For others, both the raters had good arguments for their labels, and we had to decide on that label by updating the stages in the definition of the pipeline. All the labeled pipelines from the subjects are shared in our paper artifact [23].

Furthermore, after finishing labeling the pipelines stages, we also classified the subject references into four classes based on the overall purpose of the article. First, after a few discussions, the raters and moderator came up with the classes. Then, the raters classified each pipeline into one class. We found disagreements in 6 out of 71 references, which the moderator reconciled with separate meetings with the two raters. Based on our labeling, the literature that we collected are divided into four classes: describe or propose DS pipeline, survey or review, DS optimization, and introduce new method or application. Next, we are going to discuss the result of analyzing the DS pipelines in theory.

4.2.2 Representative Pipeline in Theory

The labeled pipelines[1] with their stages are visually illustrated in the artifact [23]. We found that pipelines in theory can be both software architecture and *team processes* unlike pipelines in-the-small and in-the-large. Through the labeling process, we separated those team processes (25 out of 71), which are discussed in Section 4.2.4.

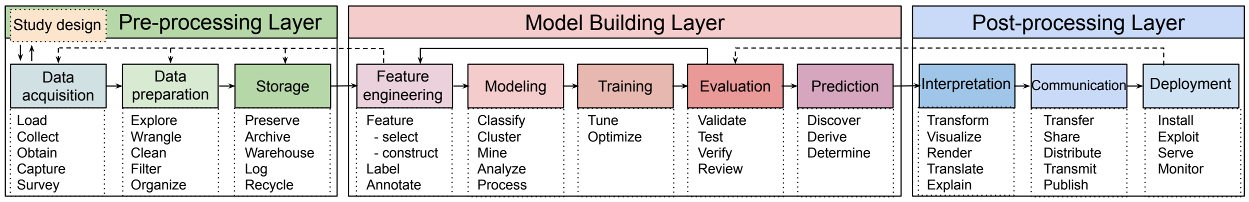

Figure 4.2: Concepts in a data science pipeline. The sub-tasks are listed below each stage. The stages are connected with feedback loops denoted with arrows. Solid arrows are always present in the lifecycle, while the dashed arrows are optional. Distant feedback loops (e.g., from *deployment* to data *acquisition*) are also possible through intermediate stage(s).

RQ1a: What is a representative definition of the DS pipeline in theory? From the empirical study, we created a representative rendition of DS pipeline with 3 layers, 11 stages and possible connections between stages as shown in Figure 4.2. Each shaded box represents a DS stage that performs certain sub-tasks (listed under the box). In the preprocessing layer, the stages are *data acquisition*, *preparation*, and *storage*. The preprocessing stage *study design* only appeared in team process pipelines that comprise requirement formulation, specification, and planning, which are often challenging in data science. The algorithmic steps and data processing are done in the model building layer. *Modeling* does not necessarily imply the existence of an ML component, since DS can involve custom data processing or statistical modeling. Post-processing layer includes the tasks that take place after the results have been generated.

RQ1b: What are the frequent and rare stages of the DS pipeline in theory? The frequency of stage can depend on the focus of the pipeline or its importance in certain context (ML, big-data management). Among 46 DS pipelines (which are not team processes), Figure 4.3 shows the number of times each stage appears. A few pipelines present stages with broad terminology that fit multiple stage-definitions. In those cases, the pipelines were labeled with the fitted stages and counted multiple times. *Modeling, data*

[1]https://github.com/sumonbis/DS-Pipeline/blob/main/pipelines.pdf

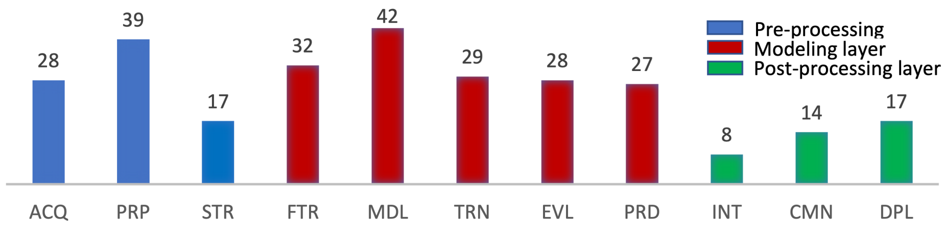

Figure 4.3: Frequency of pipeline stages in theory

preparation, and feature engineering appear most frequently in the literature. While *modeling* is present in 93% of the pipelines, other model related stages (*feature engineering, training, evaluation, prediction*) are not used consistently. Often *training* is not considered as a separate stage and included inside the *modeling* stage. Similarly, we found that *evaluation* and *prediction* are often not depicted as separate stages. However, by separating the stages and modularizing the tasks, the DS process can be maintained better [9, 173]. The pipeline created with the most number of stages (11) is provided by Ashmore et al. [12]. On the other hand, about 15% of the pipelines from the literature are created with a minimal number (3) of stages. Among them, 80% are ML processes and falls in the category of DS optimizations. We found that these pipelines are very specific to particular applications, which include context-specific stages like data sampling, querying, visualization, etc., but do not cover most of the representative stages. A pipeline in theory may not require all representative stages, since it can have novelty in certain stages and exclude the others. However, the representative pipeline provides common terminology and facilitate comparative analysis.

Finding 13: Post-processing layers are included infrequently (52%) compared to pre-processing (96%) and model building (96%) layers of pipelines in theory.

Clearly, preprocessing and model building layers are considered in almost all of the studies. In most of the cases, the pipelines do not consider the post-processing activities (*interpretation, communication, deployment*). These pipelines often end with the predictive process and thus do not follow up with the later stages which entails how the result is interpreted, communicated and deployed to the external environment. Miao et al. argued that overall lifecycle management tasks (e.g., model versioning, sharing) are largely ignored for deep learning systems [132]. Previous studies also showed that significant amount of cost and effort is spent in the post-development phases in traditional software lifecycle [129, 153]. In data-intensive software, the maintenance cost can go even higher with the high-interest technical debt in the pipeline [172]. Therefore, post-processing stages should be incorporated for a better understanding of the impact of the proposed approach on maintenance of the DS pipeline.

4.2.3 Organization of Pipeline Stages in Theory

RQ2: How are pipeline stages connected to each other? In Figure 4.2, for simplicity, we depicted the DS pipeline as a mostly linear chain. However, our subject DS pipelines often have non-linear behavior. In any stage, the system might have to return to the previous stage for refinement and upgrade, e.g., if a system faces a real-world challenge in *modeling*, it has to update the algorithm which might affect the data pre-processing and feature engineering as well. Furthermore, the stages do not have strict boundaries in the DS lifecycle. In Figure 4.2, two backward arrows, from *feature engineering* and *evaluation*, indicate feedback to any of the previous stages. Although in traditional software engineering processes (e.g., waterfall model, agile development, etc.), feedback loop is not uncommon, in DS lifecycle, there are multiple stakeholders and models in a distributed environment which makes the feedback loops more frequent and complex. Sculley et al. pointed that DS tasks such as sampling, learning, hyperparameter choice, etc. are entangled together so that Changing Anything Changes Everything (CACE principle) [173], which in turn creates implicit feedback loops that are not depicted in the pipelines [194, 45, 154, 68]. The feedback loops inside any specific layer are more frequent than the feedback loops from one layer to another. Also, a feedback loop to a distant previous stage is expensive. For example, if we do *data preparation* after *evaluation* then the intermediate stages also require updates.

4.2.4 Characteristics of the Pipelines in Theory

RQ3: What are the different types of pipelines available in theory? The context and requirements of the project can influence pipeline design and architecture [69]. Here, we present the types of pipelines with different characteristics that are available in theory. We classified each subject in our study into four classes based on the overall goal of the article. The most of the pipelines in theory (39%) are *describing or proposing* new pipelines to solve a new or existing problem. About 31% of the pipelines are on *reviewing or* *comparing* the existing pipelines. The third group of DS pipelines (14%) are intended to *optimize* a certain part of the pipeline. For example, Van Der Weide et al. proposes a pipeline for managing multiple versions of pipelines and optimize performance [194]. Most of the pipelines in this category are application specific and include very few stages that are necessary for the optimization. Fourth, some research *introduce new application* or method and present within the pipeline. We observed that there is no standard methodology to develop comparable and inter-operable DS pipelines. Using the labeling methodology shown in Figure 4.1, we labeled each pipeline and found three types of DS pipelines in the literature: 1) ML process, 2) big data management process, and 3) team process.

ML process: 46% of all the pipelines we found in the literature are describing machine learning processes. The recent advent of artificial intelligence, supervised learning and deep learning has led to more DS systems that involve ML components. The pipelines in this category emphasize the algorithmic process, learning patterns, and building predictive models. However, the post-processing stages are rare in these type of pipelines. The ML pipelines are often thought of as algorithmic process in the laboratory scenario. But as mentioned in [12], incorporating the post-processing stages would be desired to ensure safe real-world deployment of such pipelines.

Big data management: The references in this category present DS pipelines that manage a large amount of data or describes a framework (software-hardware infrastructure) for data processing but do not contain machine learning components in the pipeline. Processing large amount data often requires specific algorithms and engineering methods for efficiency and further processing. We found that 18% of all the subject studies fall in this category.

Team process: We also found some DS pipelines that are not describing DS software architecture. These pipelines describe workflow of human activities that needs to be followed in a DS pipeline. These studies present a high-level view for building DS component in a team environment. The data science teams require specific expertise and management to build successful DS pipelines [121, 9]. In this paper, in Section 4.3 and Section 4.4, we are only focusing on DS pipeline as software architecture, and therefore, we did not compare the team process pipelines in the rest of this section.

4.3 DS Pipeline in-the-Small

Similar to the DS pipelines in large systems and frameworks, for a very specific data science task (e.g., object recognition, weather forecasting, etc.), programmers build pipeline. Different stages of the program perform a specific sub-task and connect with the other stages using data-flow or control-flow relations. In this section, we described such DS pipelines *in-the-small*.

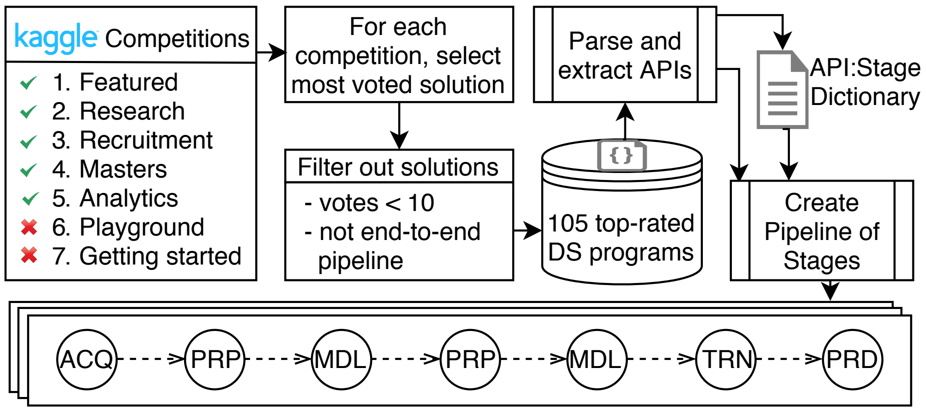

Figure 4.4: The pipeline creation process for Kaggle programs

4.3.1 Methodology

We collected 105 DS programs from Kaggle competition notebooks [99]. Kaggle is one of the most popular crowd-sourced platforms for DS competitions, owned by Google. Besides participating in competitions, data scientists, researchers, developers collaborate to learn and share DS knowledge in variety of domains. The users and organizations can host a DS competition in Kaggle to solve real-world problems. A competition is accompanied by a dataset and prize money. Many Kaggle solutions have resulted in impactful DS algorithms and research such as neural networks used by Hinton and Dahl [50], improving the search for the Higgs Boson at CERN [96], etc. We chose Kaggle solutions to analyze DS pipeline for three reasons: 1) all programs perform a DS task and provide solution to a well specified problem associated with a dataset, 2) solutions with the highest number of votes are well accepted solutions for a specific problem, and 3) the problems cover a wide range of domains.

There are 331 completed competitions in Kaggle to date. They categorized the competitions into *Featured*, *Research*, *Recruitment*, *Masters*, *Analytics*, *Playground* and *Getting started*. We collected solutions of all the competitions from each category except *Getting started* and *Playground* (these two categories are intended to serve as DS tutorials and toy projects). First, we filtered the competitions for which there are solutions available (many old competitions do not contain any public solution). We found 138 such competitions. For a given competition problem, we selected the most voted solution which has at least 10 votes. Thus, we got 105 top-rated DS solutions for analyzing pipelines in-the-small. This selection and pipeline creation process is shown in Figure 4.4.

All of the DS programs are written in Python using ML libraries like Keras, Scikit-learn, Tensorflow, etc. These packages provide high-level Application Programming Interfaces (APIs) for performing a specific task on data or model. We parsed the programs into Abstract Syntax Tree (AST) and collected all the API calls from the programs. Then the functionality of an API is used to identify the stage of the pipeline. We extracted the temporal order of API calls to identify the stages. Standard static analysis of the Python programs facilitate the extraction process. Our analysis suggests that the DS programs follow a linear structure with less than 4% AST nodes being conditional or loops. Wang et al. proposed a similar approach for extracting external dependencies in Jupyter Notebooks by creating an API database and analyzing AST [198].

We created a dictionary by mapping each API collected from the programs, to one of the 11 stages of the DS pipeline described in section Section 4.2. During the mapping, we excluded the generic APIs from the dictionary. For example, `model.summary()` is used to print the model parameters and does not represent any stage of the pipeline. For creating the dictionary, we taken a two-fold approach. First, we understand the context of the program and API usage. Second, we look at the API documentation to confirm the corresponding pipeline stage. We found that DS APIs are definitive in their operations and well-categorized by the library. For example, the APIs in Keras [119] and Scikit-learn [120] are grouped into preprocessing, models, etc. Our API-dictionary was manually validated by a second-rater and moderator who labeled DS pipelines in section Section 4.2. Then, we built a tool which takes the API dictionary and DS program, and automatically creates the DS pipeline. For a sequence of APIs with the same stage, we abstracted them into a single stage. As an example, Figure 4.4 shows a DS pipeline created from a Kaggle solution [109]. Each stage in the pipeline (e.g., ACQ, PRP) represents one or more API usages. The arrows in the pipeline denote the temporal sequence of stages. Note that, one stage can appear multiple times in a pipeline. The API dictionary, Kaggle programs, and tool to generate the pipelines is shared in the paper artifact [23].

4.3.2 Representative Pipeline in-the-Small

RQ4: What are the stages of *DS pipeline in-the-small*? Among the 11 pipeline stages described in Figure 4.2, we found only 6 stages in the DS programs that are depicted in Figure 4.5. Other stages (e.g., *storage, feature engineering, interpretation, communication, deployment*) are not found in these programs because these stages occur while building a production-scale large DS system and often not present in the DS notebooks. Therefore, the pipeline in DS programs consists of the subset of pipeline stages in theory.

Among 105 programs, *data acquisition* and *data preparation* are present in almost all of them. Surprisingly, *modeling* is present in only 70% of the programs. We found that, in many programs, no modeling APIs had been used because developers did not use any built-in ML algorithm from libraries, e.g., LogisticRegression, LSTM, etc. In these cases, the developers use data-processing APIs on the training data to build custom model, e.g., this notebook [110] uses *data preparation* APIs to produce results. To enable more abstraction of the stages in these pipelines, further modularization is necessary, which has been investigated in RQ8.

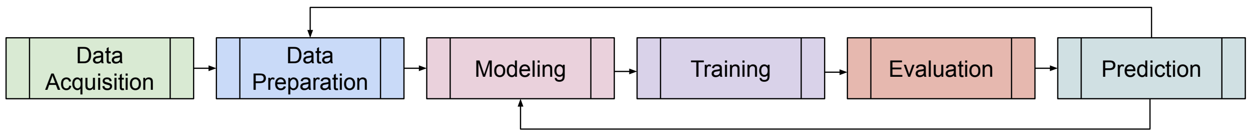

Figure 4.5: Pipeline *in-the-small* extracted from API usages

4.3.3 Pipeline Organization in the Small

RQ5: How are the stages connected with each other in pipeline in-the-small? To answer RQ5, we considered each occurrence of the stages in a DS program and looked at its previous and next stage. In Figure 4.6, we showed which stages are followed or preceded by each stage. We found that *data preparation* can occur before or after all other stages. Apart from that, *data acquisition* is followed by *data preparation*

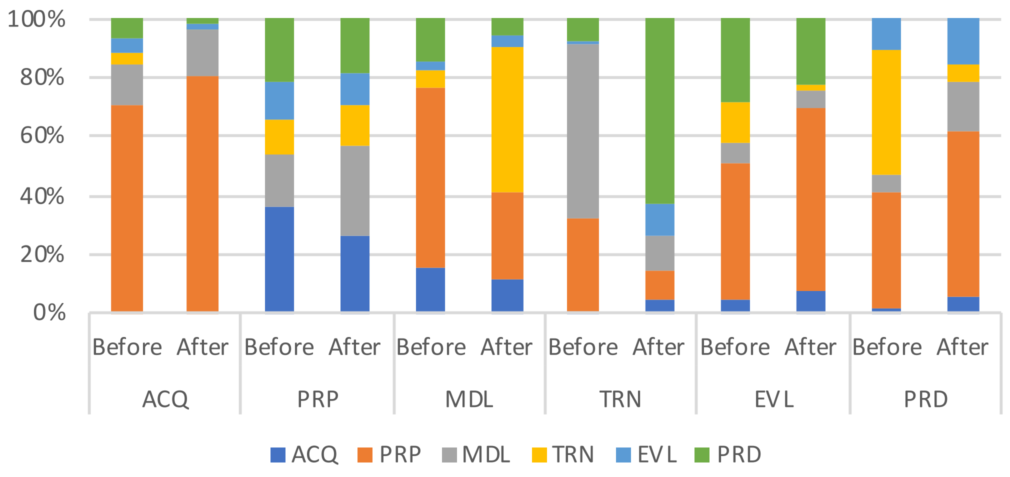

Figure 4.6: Stages occurring before and after each stage

most of the time, which in turn is followed by *modeling*. *Modeling* is followed mostly by *training*, which in turn is followed by *prediction*. *Evaluation* is mostly surrounded by *prediction* and *data preparation*. From Figure 4.6, we can also find some most occurring feedback loop: *evaluation* to *preparation*, *evaluation* to *modeling* and *prediction* to *modeling*.

Data preparation tasks (e.g., formatting, reshaping, sorting) are not limited to just before the *modeling* stage, rather it is done on a *whenever-needed* basis. For example, in the following code snippet from a Kaggle competition [111], while creating model-layers, data preprocessing API has been called in line 2.

```
1 x = Conv2D(mid, (4, 1), activation='relu', padding='valid')(x)
2 x = Reshape((branch_model.output_shape[1], mid, 1))(x)
3 x = Conv2D(1, (1, mid), activation='linear', padding='valid')(x)
4 x = Flatten(name='flatten')(x)
5 head_model = Model([xa_inp, xb_inp], x, name='head'
```

The *modeling* stage is always surrounded by other stages of the pipeline. However, there is often a loop around *modeling, training, evaluation, and prediction*. *Modeling* often repeats many times to improve the model over multiple iterations. For example, in the following Kaggle code snippet [112], the model is created and trained multiple times to find the best one.

```
# Modeling
random_forest = RandomForestClassifier(n_estimators=100, random_state=50, verbose=1, n_jobs=-1)
# Train
random_forest.fit(train, train_labels)
...
poly_features = scaler.fit_transform(poly_features)
poly_features_test = scaler.transform(poly_features_test)
```

```
random_forest_poly = RandomForestClassifier(n_estimators=100, random_state=50, verbose=1, n_jobs=-1) # Modeling
random_forest_poly.fit(poly_features, train_labels) # Training
pred = random_forest_poly.predict_proba(poly_features_test)[:,1]
```

Finding 14: Data preparation stage is occurring significant number of times between any two stages of pipelines in-the-small, which is causing pipeline jungles.

We found that new data sources are added, new features are identified, and new values are calculated incrementally in the pipeline which evolves organically. This results in a large number of data preprocessing tasks like sampling, joining, resizing along with random file input-output. This is called *pipeline jungles* [173], which causes technical debt for DS systems in the long run. *Pipeline jungles* are hard to test and any small change in the pipeline will take a lot of effort to integrate. The situation gets worse in case of larger DS pipelines, where several data management activities (e.g., clean, serve, validate) are necessary through the pipeline in different stages [149, 150]. The recommended way is to think about the pipeline holistically and scrape the pipeline jungle by redesigning it, which in turn takes further engineering effort [173]. We found that the large DS projects, which are discussed in Section 4.4, isolate the data preparation tasks into separate files and modules [33, 189, 164, 208], which alleviates the pipeline jungles problem. So, DS pipeline in-the–small needs further IDE (e.g., Jupyter Notebook, etc.) support and methodologies for code isolation and modularization.

4.3.4 Characteristics of Pipelines in-the-Small

RQ6: What are the patterns in pipeline in-the-small and how it compares to pipeline in theory? We have not found many stages from Figure 4.2, e.g., feature engineering, interpretation, communication, in pipeline in-the-small. One reason is that the low-level pipeline extracted from the API usages cannot capture some stages. For example, even if a developer is conducting feature engineering, the used APIs might be from the data preparation stage. Fortunately, we found many Kaggle notebooks that are organized by the pipeline stages. We visited all the 105 Kaggle notebooks in our collection and extracted these high-level pipelines manually. Unlike the low-level pipelines (extracted using API usages), a high-level pipeline consists of the stages abstracted by the developers.

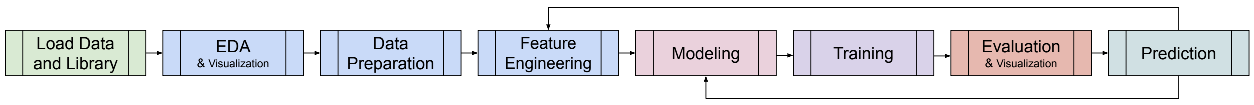

Figure 4.7: Representative data science pipeline *in-the-small*

The Kaggle notebooks follow literate programming paradigm [196, 158], which allows the developers to describe the code using rich text and separate them into sections. We found that 34 out of 105 notebooks divided the code into stages. We collected those stages from the Kaggle notebooks. Furthermore, we labeled these notebooks into the 11 stages from DS pipeline in theory by two raters, and extracted the stages that are not present in theory. The extracted high-level pipelines and labels are available in the paper artifact [23].

We observed that no notebooks specify these stages: *storage, interpretation, communication,* and *deployment*. These DS programs are not production-scale projects. Therefore, they do not include the post-processing stages in the pipeline. The most common stages are *modeling* (79%), *data preparation* (62%), *data acquisition* (53%), and *feature engineering*(35%), which is aligned with the finding of DS pipeline in theory. In addition, we found these stages which are not present in theory: *library loading, exploratory data analysis (EDA), visualization*. Among them *EDA* has been used most of the times (43%) and covered the most part of those pipeline. Before going to the modeling and successive stages, a lot of effort is given on understanding the data, compute feature importance, and visualize the patterns, which help to build models quickly in later stages [12].

Furthermore, some notebooks present *library loading* as separate stage. We observed that choosing appropriate library/framework and setting up the environment is an important step while developing pipeline in-the-small. We also found that data visualization is an recurring stage mentioned by the developers. Visualization can be done for EDA or feature engineering (before modeling), or for evaluation (after modeling). Based on these observations we updated the representative pipeline in-the-small in Figure 4.7. The high-level pipeline provides an overall representation of the system, which can be leveraged to design software process. It would be beneficial for the developers to close the gap between the low-level and the high-level pipeline by identifying the tangled stages.

4.4 DS Pipeline in-the-Large

The DS solutions described in the previous section are specific to a given dataset and a well-defined problem. However, there are many DS projects which are large, not limited to a single source file, and contains multiple modules. These solutions are intended to solve more general problems which might not be specific to a dataset. For example, the objective of the *Face Classification* project in GitHub [11] is to detect face from images or videos and classify them based on gender and emotion. This problem is not specific to a particular dataset and the scope is broader compared to the Kaggle solutions. We collected such top-rated DS projects from GitHub to analyze DS pipeline in-the-large.

Table 4.2: GitHub projects for analyzing pipeline in-the-large

Project Name	Purpose	#Files	#AST	LOC
Autopilot [7]	Pilot a car using computer vision	36	11185	348
CNN-Text-Classification [33]	Sentence classification	69	47797	11.4K
Darkflow [189]	Real-time object detection and classification	1025	655670	8.6K
Deep ANPR [60]	Automatic number plate recognition	64	70464	10.8K
Deep Text Corrector [139]	Correct input errors in short text	47	50770	3.0K
Face Classification [11]	Real-time face and emotion/gender detection	292	117901	35.3K
FaceNet [164]	Face recognition	1352	1889529	18.2K
KittiSeg [185]	Road segmentation	276	187143	4.8K
LSTM-Neural-Network [14]	Predict time series steps and sequences	24	11434	1.2K
Mask R-CNN [2]	Object detection and instance segmentation	256	1567786	15.6K
MobileNet SSD [152]	Object detection network	28	21272	25.6K
MTCNN [47]	Joint face detection and alignment	153	121138	219.7K
Object-Detector-App [52]	Real-time object recognition	215	318534	47.9K
Password-Analysis [161]	Analyze a large corpus of text passwords	148	67870	3.6K
Person Blocker [205]	Block people in images	12	44517	977
QANet [208]	Machine reading comprehension	83	107669	2K
Speech-to-Text-WaveNet	Sentence level english speech recognition	32	18626	5.1K
Tacotron [141]	Text-to-speech synthesis	114	58845	1.4K
Text-Detection-CTPN [157]	Text detection	640	257083	18.4K
TF-Recomm [180]	Recommendation systems	17	7789	535
XLNet [219]	Language understanding	36	143172	11.5K

4.4.1 Methodology

Biswas et al. published a dataset containing top rated DS projects from GitHub [25]. From the list of projects in this dataset, we filtered mature DS projects having more than 1000 stars. Thus, we found 269 mature GitHub projects. However, there are many projects in this list which are DS libraries, frameworks or utilities. Since we want to analyze the pipeline of data science software, we removed those projects. Finally, we also removed the repositories which serve educational purposes. Thus, we found a list of 21 mature open-source DS projects. The list of projects, and their purpose are shown in Table 4.2.

For each project, we created two pipelines: high-level pipeline and low-level pipeline. For creating the high-level pipeline, we manually checked the project architecture, module structure and execution process. This gave us a good understanding of the source file organization and linkage between modules. After identifying the high-level pipeline and execution sequences of the source files, we used the same API based method used to analyze Kaggle programs in the previous section, to create low-level pipeline of these GitHub projects. The methodology of selecting and extracting pipelines from the GitHub projects is shown in Figure 4.8.

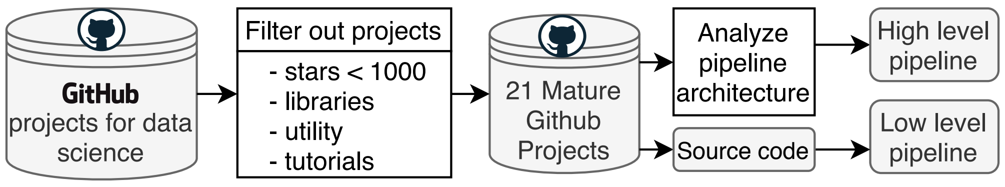

Figure 4.8: Pipeline creation process for GitHub projects

For example, the project *QANet* [208] is intended to do machine reading comprehension. Here, Python has been used as the primary language, and shell script has been used for data downloading and project setup. The high-level pipeline for *QANet* includes the stages: *data acquisition*, *data preparation*, *modeling*, *training*, *evaluation* and *prediction*. In the beginning, `config.py` file integrates the modules (preparation, modeling, and training) and provides an interface to configure a model by specifying dataset and other parameters. Then, the file `evaluate.py` is executed to perform the evaluation and prediction. For the low-level pipeline, for a specific file, we used the API based analysis to generate the pipeline, which was used to analyze pipeline in-the-small. For instance, in the project *QANet*, although `model.py` serves modeling at a high level, it also does data preparation, training, and evaluation, when APIs are considered. In addition to the pipeline stages, we also identified a few other properties of each project: 1) number of contributors, 2) AST count, 2) technology/language used, 3) entry points and 4) execution sequence. We leveraged the Boa infrastructure [58, 59] to analyze the different properties of the projects. These properties helped us to categorize and analyze the pipeline in-the-large. The details of the projects are available in the paper artifact [23].

The projects are from various domains: object detection, face classification, automated driving, speech synthesis, number plate recognition, predict time series sequence, etc. The number of developers in each project ranges between 1 and 40 with an average of 8. Among 21 projects, 16 of them are developed by teams and 5 of them are developed by individuals. The primary language used to develop these projects is Python.

4.4.2 Representative Pipeline in-the-Large

Compared to the Kaggle programs, we found a significant difference in the pipeline of large DS projects. Because of the larger size of the projects, the pipeline architecture is different. All the projects contain multiple source files for handling different tasks (e.g., modeling, training) and about 50% of the projects organize the source files into modules (e.g., *utils*, *preprocessing*, *model*, etc.).

RQ7: What is the representative DS pipeline in-the-large? Each of the projects contains six stages described in Figure 4.5: *acquisition, preparation, modeling, training, evaluation*, and *prediction*. However, since the projects are not coupled to a specific dataset and they solve a more general problem, the projects are not limited to one single pipeline. We found that the pipeline of each project is divided into two phases: 1) development phase and 2) post-development phase, which is depicted in Figure 4.9.

In **development phase**, the main goal is to build a model that solves the problem in general. A base dataset is used to build the model that would be used for other future datasets. After completing a *modeling, training, evaluation* loop, the final model is created and saved as an artifact. Afterwards, the projects also create model interfaces, which lets the user modify and exploit the model in the post-development phase. Finally, the model artifact is saved as a source file or some model archiving formats. For example, the project *Person-Blocker* [205] and *Speech-to-Text-WaveNet* [122] saved the model in the source file (`model.py`) and

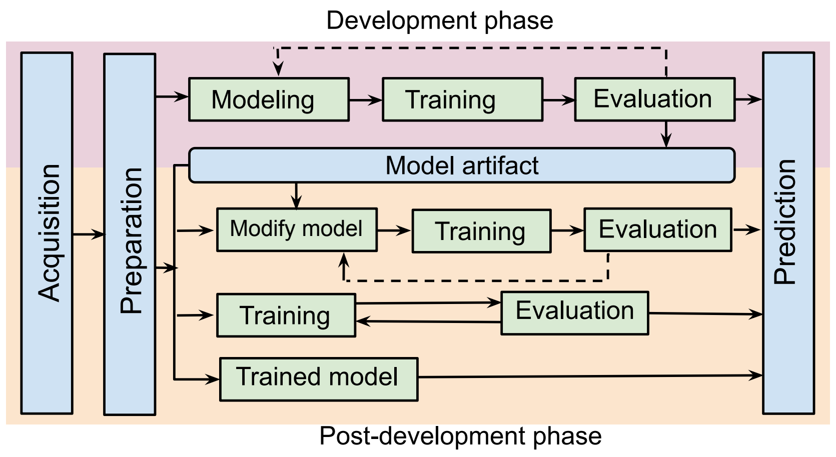

Figure 4.9: Data science pipeline in-the-large. Development phase (top) runs during model building and post-development phase (bottom) runs for making prediction. In development phase, a model is created as an artifact. In post-development phase, the model artifact is used for prediction on new dataset. There can be three different patterns in the post-development phase.

lets the users train the model in the next phase. On the other hand, the project *KittiSeg* [185] and *Autopilot* [7] saved the built model artifact in JSON format (`.json`) and checkpoint format (`.ckpt`) respectively. We observed that the evaluation and prediction is often not the main goal in this phase; rather, building an appropriate model and making it available for further usage is the central activity.

In **post-development phase**, the users access the pre-built model and use that for prediction. After acquiring data, a few preprocessing steps are needed to feed the model. In all of the projects under this study, we found that the development phase is similar. However, we identified three different patterns in the post-development phase which are shown in Figure 4.9. First, the users can modify the model by setting its hyperparameters and use that to make prediction on a new dataset. Second, the users can use the model as-it-is and train the model on the new dataset to make prediction. Third, the users can also download the pre-trained model and directly leverage that for prediction. Finally, at the end of this phase, the prediction result is obtained.

The post-development phase in the pipeline enabled software reusability of the models. All of these projects have instructions in their `readme` or documentation explaining the usage and customization. For example, the project Deep ANPR [60] provides instructions for obtaining large training data, retraining the models, and build it for prediction. However, not all the projects enable reusability in the development pipelines. Only a few of them provides access to the modules by importing in new development scenario. For instance, Darkflow [189] let users access the `darkflow.net.build` module and use it in new application development. To increase the reusability of DS programs, it would be desired to consider similar access to the development pipeline of these large projects.

4.4.3 Organization of DS Pipeline in-the-Large

RQ8: How are the stages connected in pipeline in-the-large? The abstraction in DS projects is stricter than the DS programs described in Section 4.3. The projects are built in a modular fashion, i.e., one source file for a broad task (e.g., `train.py, model.py`). However, inside one specific file, there are many other possible stages, especially data *preprocessing* appears inside all the source files. In addition, the module connectivity is not linear. All of the modules use external libraries for performing different tasks. As a result, there are a lot of interdependencies (both internal and external) in the DS pipeline. One immediate difference of these pipelines with traditional software is DS pipelines are heavily dependant on the data. For example, the project *Speech-to-Text-WaveNet* [122] requires a certain format of data. When we want to use that in a new situation, the data properties might be different. So, the usage pipelines would have a few additional stages. In some cases, the original pipeline is modified. Here, there are many sub-pipelines work together to build a large pipeline. However, we have not found any framework or common methodology these software are using. The different patterns of DS pipelines seek more advanced methodology or framework to build DS pipeline and release for production.

4.4.4 Characteristics of Pipelines in-the-Large

RQ9: What are the patterns found in the pipelines? The pipelines found in this setting can be categorized into 1) loosely coupled and 2) tightly coupled, based on their modularity. A high number of contributors in the project resulted in loosely coupled pipelines. We found the loosely coupled ones are designed in a modular fashion and one module (e.g., data cleaning, modeling) is designed to be used by other modules. Usually, there are multiple entry-points in a loosely coupled pipeline and user has more flexibility. On the other hand, in a tightly coupled pipeline, the modules are stricter and integrated tightly with other modules. There is only one or two entry-points to the pipeline, which automatically calls the other modules. We found that the projects with 6 or more contributors ($\sim$75%) followed a loosely coupled architecture and projects with 1 to 5 contributors followed a tightly coupled architecture.

4.5 Summary

Many software systems today are incorporating a data science pipeline as their integral part. In this work, we argued that to facilitate research and practice on data science pipelines, it is essential to understand their nature. To that end, we presented a three-pronged comprehensive study of data science pipelines in theory, data science pipelines in-the-small, and data science pipelines in-the-large. Our study analyzed three datasets: a collection of 71 proposals for data science pipelines and related concepts in theory, a collection of 105 implementations of data science pipelines from Kaggle competitions to understand data science in-the-small, and a collection of 21 mature data science projects from GitHub to understand data science in-the-large. We have found that DS pipelines differ significantly between these settings. Specifically, a number of stages are absent in-the-small, and the DS pipelines have a more linear structure. The DS pipelines in-the-large have a more complex structure and feedback loops compared to the theoretical representations. We also contribute three representations of DS pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large.

CHAPTER 5. FAIR PREPROCESSING: TOWARDS UNDERSTANDING COMPOSITIONAL FAIRNESS OF DATA TRANSFORMERS IN MACHINE LEARNING PIPELINE

In recent years, many incidents have been reported where machine learning models exhibited discrimination among people based on race, sex, age, etc. Research has been conducted to measure and mitigate unfairness in machine learning models. For a machine learning task, it is a common practice to build a pipeline that includes an ordered set of data preprocessing stages followed by a classifier. However, most of the research on fairness has considered a single classifier based prediction task. What are the fairness impacts of the preprocessing stages in machine learning pipeline? Furthermore, studies showed that often the root cause of unfairness is ingrained in the data itself, rather than the model. But no research has been conducted to measure the unfairness caused by a specific transformation made in the data preprocessing stage. In this chapter, we introduced the causal method of fairness to reason about the fairness impact of data preprocessing stages in ML pipeline. We leveraged existing metrics to define the fairness measures of the stages. Then we conducted a detailed fairness evaluation of the preprocessing stages in pipelines collected from three different sources. Our results show that certain data transformers are causing the model to exhibit unfairness. We identified a number of fairness patterns in several categories of data transformers. Finally, we showed how the local fairness of a preprocessing stage composes in the global fairness of the pipeline. We used the fairness composition to choose appropriate downstream transformer that mitigates unfairness in the machine learning pipeline.

5.1 Introduction

Fairness of machine learning (ML) predictions is becoming more important with the rapid increase of ML software usage in important decision making [54, 136, 10, 74], and the black-box nature of ML algorithms [66, 5]. There is a rich body of work on measuring fairness of ML models

[210, 56, 64, 84, 36, 48, 212, 181] and mitigate the bias [213, 36, 210, 48, 73, 84, 148, 114]. Recent work [34, 88, 65, 27, 85, 42] has shown that more software engineering effort is required towards detecting bias in complex environment and support developers in building fairer models.

The majority of work on ML fairness has focused on classification task with single classifier [66, 64, 5, 32]. However, real-world machine learning software operate in a complex environment [32, 51]. In an ML task, the prediction is made after going through a series of stages such as data cleaning, feature engineering, etc., which build the machine learning pipeline [8, 207]. Studying only the fairness of the classifiers (e.g., *Decision Tree*, *Logistic Regression*) fails to capture the fairness impact made by other stages in ML pipeline. In this paper, we conducted a detailed analysis on how the data preprocessing stages affect fairness in ML pipelines.

Prior research observed that bias can be encoded in the data itself and missing the opportunity to detect bias in earlier stage of ML pipeline can make it difficult to achieve fairness algorithmically [123, 88, 81, 54]. Additionally, bias mitigation algorithms operating in the preprocessing stage were shown to be successful [113, 64]. Therefore, it is evident that the preprocessing stages of ML pipeline can introduce bias. However, no study has been conducted to measure the fairness of the preprocessing stages and show how it impacts the overall fairness of the pipeline. In this paper, we used the causal method of fairness to reason about the fairness impact of preprocessing stages in ML pipeline. Then, we leveraged existing fairness metrics to measure fairness of the preprocessing stages. Using the measures, we conducted a thorough analysis on a benchmark of 37 real-world ML pipelines collected from three different sources, which operate on five datasets. These ML pipelines allowed us to evaluate fairness of a wide selection of preprocessing stages from different categories such as data standardization, feature selection, encoding, over/under-sampling, imputation, etc. For comparative analysis, we also collected data transformers e.g., `StandardScaler`, `MinMaxScaler`, `PCA`, `l1-normalizer`, `QuantileTransformer`, etc., from the pipelines as well as corresponding ML libraries, and evaluated fairness. Finally, we investigated how fairness of these preprocessing techniques (*local fairness*) composes with other preprocessing stages, and the whole pipeline (*global fairness*). Specifically, we answered the following three research questions.

RQ1 (fairness of preprocessing stages): What are the fairness measures of each preprocessing stage in ML pipeline? **RQ2** (fair transformers): What are the fair (and biased) data transformers among the commonly used ones? **RQ3** (fairness composition): How fairness of data preprocessing stages composes in ML pipeline?

- How local fairness compose into global fairness?
- Does choosing a downstream transformer depend on the fairness of an upstream transformer?

To the best of our knowledge, we are the first to evaluate the fairness of preprocessing stages in ML pipeline. Our results show that by measuring the fairness impact of the stages, the developers would be able to build fairer predictions effectively. Furthermore, the libraries can provide fairness monitoring into the data transformers, similar to the performance monitoring for the classifiers. Our evaluation on real-world ML pipelines also suggests opportunities to build automated tool to detect unfairness in the preprocessing stages, and instrument those stages to mitigate bias. We have made the following contributions in this paper:

1. We created a fairness benchmark of ML pipelines with several stages. The benchmark, code and results are shared in our replication package[1] in GitHub repository, that can be leveraged in further research on building fair ML pipeline [26].
2. We introduced the notion of causality in ML pipeline and leveraged existing metrics to measure the fairness of preprocessing stages in ML pipeline.
3. Unfairness patterns have been identified for a number of stages. These stages should be used cautiously in the pipelines.
4. We identified alternative data transformers which can mitigate bias in the pipeline.
5. Finally, we showed the composition of stage-specific fairness into overall fairness, which is used to choose appropriate downstream transformer that mitigates bias.

[1]https://github.com/sumonbis/FairPreprocessing

The paper is organized as follows: Section 5.2 describes the motivating examples, Section 5.3 describes the existing metrics and our approach. In Section 5.4, we described the benchmark and experiments. Section 5.5 explores the results, Section 5.6 provides a comparative study among transformers, and Section 5.7 evaluates the fairness composition. Finally, Section 5.8 describes the threats to validity, Section 5.9 discusses related work, and Section 5.10 concludes.

5.2 Motivation

In this section, we present two ML pipelines which show that the preprocessing stage affects the fairness of the model and it is important to study the bias induced by certain data transformers.

5.2.1 Motivating Example 1

Yang *et al.* [207] studied the following ML pipeline which was originally outlined by Propublica for recidivism prediction on *Compas* dataset [10]. The goal is predict future crimes based on the data of defendants. The fairness values, in terms of statistical parity difference (SPD: -0.102) and equal opportunity difference (EOD: -0.027), suggest that the prediction is biased towards[2] *Caucasian* defendants when *race* is considered as sensitive attribute. The pipeline consists of several preprocessing stages before applying `LogisticRegression` classifier. Data preprocessing includes cleaning, encoding categorical features, and missing value imputation. Recent research [207] showed that the data transformation in this pipeline is not symmetric across gender groups i.e., male defendants are filtered more than the female. Do these data transformations introduce unfairness in the prediction? If yes, what are the unfairness measures of these transformers? Is it possible to leverage existing metrics to measure the unfairness of each component? If we can understand the effect of each data transformer, it would be possible to choose data preprocessing technique wisely to avoid introducing bias as well as mitigate the inherent bias in data or classifier.

```
df = pd.read_csv(f_path)
df = df[(df.days_b_screening_arrest <= 30) & (df.days_b_screening_arrest >= -30)
    & (df.is_recid != -1) & (df.c_charge_degree != 'O') & (df.score_text != 'N/A')]
df = df.replace('Medium', 'Low')
labels = LabelEncoder().fit_transform(df.score_text)
impute1_onehot = Pipeline([ ('imputer1', SimpleImputer(strategy='most_frequent')),
```

[2] Bias *towards* a group connotes that the prediction favours that group.

```
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])
impute2_bin = Pipeline([('imputer2', SimpleImputer(strategy='mean')),
    ('discretizer', KBinsDiscretizer(n_bins=4, encode='ordinal', strategy='uniform'))])
featurizer = ColumnTransformer(transformers=[('impute1_onehot', impute1_onehot, ['is_recid']),
    ('impute2_bin', impute2_bin, ['age'])])
pipeline = Pipeline([('features', featurizer),
    ('classifier', LogisticRegression())])
```

5.2.2 Motivating Example 2

The following ML pipeline is collected from the benchmark used by Biswas and Rajan [27] for studying fairness of ML models. This pipeline operates on *German Credit* dataset. Here, the goal is to predict the credit risk (good/bad) of individuals based on their personal data such as age, sex, income, etc. In this pipeline, before training the classifier, data has been processed using two transformers: `PCA` for principal component analysis, and `SelectKBest` for selecting high-scoring features. The fairness value (SPD: 0.005) shows that prediction is slightly biased towards *female* candidates. However, if the transformers are not applied, then prediction becomes biased towards *male* (SPD: -0.117). By applying one transformer at a time, we observed that `PCA` alone is not causing the change of fairness. In this case, `SelectBest` is causing bias towards *female*, which in turn mitigating the overall fairness of the pipeline. Therefore, in addition to study the fairness of transformers in isolation, it is important to understand how fairness of components composes in the pipeline.

```
features = []
features.append(('pca', PCA(n_components=2)))
features.append(('select_best', SelectKBest(6)))
feature_union = FeatureUnion(features)
estimators = []
estimators.append(('feature_union', feature_union))
estimators.append(('RF', RandomForestClassifier()))
model = Pipeline(estimators)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Our key idea is to leverage causal reasoning and observe fairness impact of a stage on prediction. To do that we create alternative pipeline by removing a stage. For example, from the above pipeline, we remove the `SelectKBest` and compare the predictions with original pipeline. We observe that `SelectKBest` is causing 1.1% of the female and 3.6% of the male participants to change predictions from favorable (good credit) to unfavorable (bad credit). Since the stage is causing more unfavorable decisions to male, the stage is biased towards female. Thus, we used existing fairness criteria to measure fairness impact of a stage and propose novel metrics.

5.3 Methodology

In this section, first, we describe the background of ML pipeline, focussing on the data preprocessing stages. Second, we formulate the method and metrics to measure fairness of a certain preprocessing stage with respect to the pipeline it is used within.

5.3.1 ML Pipeline

Amershi *et al.* proposed a nine-stage machine learning pipeline with data-oriented (collection, cleaning, and labeling) and model-oriented (model requirements, feature engineering, training, evaluation, deployment, and monitoring) stages [8]. Other research [1, 18] also described data preprocessing as an integral part of the ML pipeline. In Chapter 4, we have done a comprehensive study on the organization and characteristics of the pipelines. After acquiring raw data, each of these pipelines goes through a set of data preprocessing stages. Finally, using the processed data, a classifier (or an ensemble) is trained to build the model, which is used to make prediction on unseen data. The pipelines in the motivating examples are depicted in Figure 5.1, which follows the representation provided by Yang *et al.* [207]. In this paper, we adapted the canonical definition of pipeline from Scikit-Learn pipeline specification [35, 171], which is aligned with the ML models studied in the literature for fair classification tasks [19, 207, 27, 65, 5, 66]. We are interested in investigating the fairness of the data preprocessing stages in the pipeline, which is depicted with grey boxes in Figure 5.1.

To summarize, a canonical *ML pipeline* is an ordered set of m stages with a set of *preprocessing stages* $(S_1, S_2, \ldots S_{m-1})$ and a final *classifier* (S_m). Each preprocessing stage, S_k operates on the data already processed by preceding stages $S_1, \ldots S_{k-1}$. A data preprocessing stage S_k can be a data transformer or a set of custom operations. A *data transformer* is a well-known algorithm or method to perform a specific operation such as variable encoding, feature selection, feature extraction, dimensionality reduction, etc. on the data [35]. For example, in the second motivating example, two transformers (`PCA` and `SelectKBest`) have been used. *Custom transformation* includes data/task-specific contextual operations on the dataset. For

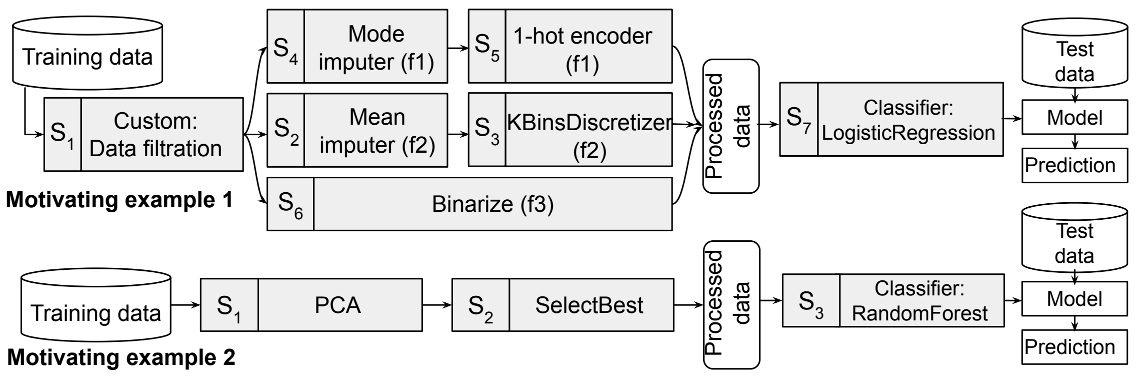

Figure 5.1: Machine learning pipelines for the motivating examples, having a sequence of preprocessing stages followed by a classifier. In the first pipeline, the set of stages $\{S_2, S_3\}, \{S_4, S_5\}, \{S_6\}$ can be executed sequentially or parallel, since they operate on different features: *f1*, *f2*, *f3*. Otherwise, maintaining the order of stages is necessary.

example, in Section 5.2.1 (line 2-3), the data instances that do not contain a value in the range [-30, 30] for the feature `days_b_screening_arrest`, have been filtered. This means the pipeline ignored the data of the defendants with more than 30 days between their screening and arrest. This formulation of ML pipeline allowed us to evaluate fairness of the preprocessing stages in real-world ML tasks.

5.3.2 Existing Fairness Metrics

We have leveraged existing fairness metrics to measure the fairness of the whole pipeline. Many fairness metrics have been proposed in the literature for measuring fairness of classification tasks [19, 22, 54]. In general, the fairness metrics compute group-specific classification rates (e.g., true positives, false positives), and calculates the difference between groups to measure the fairness. In this paper, we adopted the representative group fairness metrics used by [65, 27]. Specifically, we leveraged the following metrics: statistical parity difference (SPD) [113, 210, 64], equal opportunity difference (EOD) [84], average odds difference (AOD) [84], and error rate difference (ERD) [48]. Given a dataset D with n instances, let, actual classification label be Y, predicted classification label be $\hat{Y}$, and sensitive attribute be $\mathcal{A}$. Here, $Y = 1$ if the label is favorable to the individuals, otherwise $Y = 0$. For example, classification task on *German Credit* dataset predicts the credit risk (good/bad credit) of individuals. In this case, $Y = 1$ if the prediction is *good credit*, otherwise $Y = 0$. Suppose, for privileged group (e.g., *White*), $\mathcal{A} = 1$ and for unprivileged group (e.g., *non-White*), $\mathcal{A} = 0$. SPD is computed by observing the probability of giving favorable label to each group and taking the difference. EOD measures the true-positive rate difference between groups. AOD calculates both true positive rate and false positive rate difference and then takes the average. ERD calculates the sum of false positive rate difference and false negative rate difference between groups. The definitions of these metrics are as follows:

$$\begin{aligned}
\mathsf{SPD} &= \mathsf{P}[\hat{Y} = 1|\mathcal{A} = 0] - \mathsf{P}[\hat{Y} = 1|\mathcal{A} = 1] \\
\mathsf{EOD} &= \mathsf{P}[\hat{Y} = 1|Y = 1, \mathcal{A} = 0] - \mathsf{P}[\hat{Y} = 1|Y = 1, \mathcal{A} = 1] \\
\mathsf{AOD} &= (1/2)\{(\mathsf{P}[\hat{Y} = 1|Y = 1, \mathcal{A} = 0] - \mathsf{P}[\hat{Y} = 1|Y = 1, \mathcal{A} = 1]) \\
&\quad + (\mathsf{P}[\hat{Y} = 1|Y = 0, \mathcal{A} = 0] - \mathsf{P}[\hat{Y} = 1|Y = 0, \mathcal{A} = 1])\} \\
\mathsf{ERD} &= (\mathsf{P}[\hat{Y} = 1|Y = 0, \mathcal{A} = 0] - \mathsf{P}[\hat{Y} = 1|Y = 0, \mathcal{A} = 1])\} \\
&\quad + (\mathsf{P}[\hat{Y} = 0|Y = 1, \mathcal{A} = 0] - \mathsf{P}[\hat{Y} = 0|Y = 1, \mathcal{A} = 1])\}
\end{aligned} \tag{5.1}$$

Disparate impact (DI) and statistical parity difference (SPD) both measure the same rate i.e., probability of classifying data instance as favorable, but DI computes the ratio of privileged and unprivileged groups' rate, whereas SPD computes the difference. Therefore, from DI and SPD, we only used SPD in our evaluation.

5.3.3 Fairness of Preprocessing Stages

Suppose, $\mathcal{P}$ is a pipeline with m stages and our goal is to evaluate the fairness of the stage S_k, where $1 \leq k < m$. In other words, we want to measure the fairness impact of S_k on the prediction made by $\mathcal{P}$. To achieve that we applied the causal reasoning for evaluating fairness. The causality theorem was proposed by Pearl [144, 145] and further studied extensively to reason about fairness in many scenarios [66, 127, 215, 163, 160]. Causality notion of fairness captures that everything else being equal, the prediction would not be changed in the counterfactual world where only an intervention happens on a variable [66, 127, 160]. For example, Galhotra et al. proposed causal discrimination score for fairness testing [66]. The authors created test inputs by altering original protected attribute values of each data instance, and observed whether prediction is changed for those test inputs. If the intervention causes the prediction to be changed, we call the software causally unfair with respect to that intervention. In our case, if a preprocessing stage S_k be the intervention, to measure the fairness of S_k, we have to capture the prediction disparity caused by the intervention S_k. This causal reasoning of fairness is a stronger notion since it provides causality in software by observing changes in the outcome made by a specific stage in the pipeline [66, 145].

5.3.3.1 Causal Method to Measure Fairness of Preprocessing Stage

From pipeline $\mathcal{P}$, we construct another pipeline $\mathcal{P}^*$ by only excluding the stage S_k from $\mathcal{P}$. After applying the stage S_k in $\mathcal{P}^*$, to what extent the prediction of $\mathcal{P}^*$ changes, and whether the change is favorable to any group? Broadly, this can be measured by observing the prediction difference between $\mathcal{P}$ and $\mathcal{P}^*$ and computing the fairness of these changes using the fairness metrics from Equation (5.1).

Suppose, the predictions made by the two pipelines are $\hat{Y}(\mathcal{P})$ and $\hat{Y}(\mathcal{P}^*)$. Let, I be the impact set for S_k, which denotes the prediction parity between $\hat{Y}(\mathcal{P})$ and $\hat{Y}(\mathcal{P}^*)$ such that for i^{th} data instance, if $\hat{Y}_i(\mathcal{P}) = \hat{Y}_i(\mathcal{P}^*)$, then $I_i = 0$, otherwise 1. By causality, the fairness of preprocessing stage (denoted by SF) is calculated based on $[\hat{Y}(\mathcal{P}), \hat{Y}(\mathcal{P}^*)]_{I=1}$ with respect to a fairness metric M, which is shown in Equation (5.2a). We noticed that a few preprocessing stages, specifically the encoders can not be removed without replacing with an alternative stage. For such situations, we have defined the fairness of S_k with reference to another stage S'_k, denoted by $\mathsf{SF(S_k|S'_k)}$ in Equation (5.2b).

Zelaya also used the similar method for quantifying the effect of a preprocessing stage with a goal of computing *volatility* of a stage [211]. *Volatility* quantifies how much impact a preprocessing stage has on the outcome by computing the probability of prediction changes. However, it does not capture the fairness of the stage, since a stage can cause high change in the prediction by maintaining the predictions fair. Next in Section 5.3.3.2, we have extended our causality based formulation of Equation (5.2a) for each fairness metric in Equation (5.1) to capture the fairness impact of each preprocessing stage. Similar to [66], the benefit of this formulation is, the measures do not require an oracle, since the prediction equivalence of pipelines $\mathcal{P}$ and $\mathcal{P}^*$ serves the goal of evaluating fairness of the stage. Note that the rest of the definitions in Section 5.3.3.2 are independent of Equation (5.2a) and Equation (5.2b).

$$I = \begin{cases} 0 & \text{if } \hat{Y}_i(\mathcal{P}) = \hat{Y}_i(\mathcal{P}^*) \\ 1 & \text{otherwise} \end{cases}, \text{ for all } i \in \{1 \ldots n\}$$

$$\mathsf{SF}(S_k) = M[\hat{Y}(P), \hat{Y}(P^*)]_{I=1} \text{ where } \mathcal{P}^* = \mathcal{P} \setminus S_k \tag{5.2a}$$

$$\mathsf{SF}(S_k | S'_k) = M[\hat{Y}(P), \hat{Y}(P^*)]_{I=1} \text{ where } \mathcal{P}^* = (\mathcal{P} \setminus S_k) \cup S'_k \tag{5.2b}$$

5.3.3.2 Fairness Metircs for Preprocessing Stage

We have leveraged the definition of metrics SPD, EOD, AOD, and ERD from Equation (5.1) to capture the stage-specific fairness of S_k. Essentially, the new metrics will identify the disparities between $\hat{Y}_i(\mathcal{P})$ and $\hat{Y}_i(\mathcal{P}^*)$ and use corresponding fairness criteria to measure how much S_k favors a specific group with respect to other group(s).

Suppose, among n data instances, n_u are from the unprivileged group and n_p from the privileged group. $\mathsf{SFC}_{\mathsf{SPD}}$ computes how many of the data instances have been changed from unfavorable to favorable after applying the stage S_k. To do that we count changes in both directions (unfavorable to favorable and favorable to unfavorable), and take the difference. The sign of $\mathsf{SFC}_{\mathsf{SPD}}$ preserves the direction of changes. Finally, the metric $\mathsf{SF}_{\mathsf{SPD}}$ is computed by taking the difference of rates ($\mathsf{SFR}_{\mathsf{SPD}}$) between unprivileged and privileged groups. Note that the metric captures fairness by measuring the difference of favorable change rates between groups. Simply counting the mismatches between $\hat{Y}_i(\mathcal{P})$ and $\hat{Y}_i(\mathcal{P}^*)$ could provide degree of changes in $\mathsf{SFC}_{\mathsf{SPD}}$ but would not capture fairness. Furthermore, computing favorable changes to both groups separately and evaluating the disparity between them captures fairness according to the original definition of SPD.

$$\mathsf{SFC}_{i\mathsf{SPD}} = \begin{cases} 1 & \text{if } \hat{Y}_i(\mathcal{P}) = 1 \text{ and } \hat{Y}_i(\mathcal{P}^*) = 0 \\ -1 & \text{if } \hat{Y}_i(\mathcal{P}) = 0 \text{ and } \hat{Y}_i(\mathcal{P}^*) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{SFC}_{\mathsf{SPD}} = \sum_{i=1}^{n} \mathsf{SFC}_{i\mathsf{SPD}}$$

$$\mathsf{SFR}_{\mathsf{SPD}}(u) = \mathsf{SFC}_{\mathsf{SFD}}(u)/n_u,\ \mathsf{SFR}_{\mathsf{SPD}}(p) = \mathsf{SFC}_{\mathsf{SPD}}(p)/n_p$$

$$\mathsf{SF}_{\mathsf{SPD}} = \mathsf{SFR}_{\mathsf{SPD}}(u) - \mathsf{SFR}_{\mathsf{SPD}}(p) \tag{5.3}$$

Similarly, $\mathsf{SF}_{\mathsf{EOD}}$ is defined using the following equation. In this case, only the true-positive changes are considered as suggested by the definition of EOD from Equation (5.1).

$$\mathsf{SFC}_{i\mathsf{EOD}} = \begin{cases} 1 & \text{if } Y_i = \hat{Y}_i(\mathcal{P}) = 1 \text{ and } \hat{Y}_i(\mathcal{P}^*) = 0 \\ -1 & \text{if } \hat{Y}_i(\mathcal{P}) = 0 \text{ and } Y_i = \hat{Y}_i(\mathcal{P}^*) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{SFC}_{\mathsf{EOD}} = \sum_{i=1}^{n} \mathsf{SFC}_{i\mathsf{EOD}}$$

$$\mathsf{SFR}_{\mathsf{EOD}}(u) = \mathsf{SFC}_{\mathsf{EOD}}(u)/n_{u,Y=1},\ \mathsf{SFR}_{\mathsf{EOD}}(p) = \mathsf{SFC}_{\mathsf{EOD}}(p)/n_{p,Y=1}$$

$$\mathsf{SF}_{\mathsf{EOD}} = \mathsf{SFR}_{\mathsf{EOD}}(u) - \mathsf{SFR}_{\mathsf{EOD}}(p) \tag{5.4}$$

Since AOD computes the average of true positive (TP) rate and false positive (FP) rate, first the change set for TP and FP predictions is computed. Then averaging the probability of changes for TP and FP, the change rates are computed for both groups. Finally, $\mathsf{SF}_{\mathsf{AOD}}$ is calculated by taking the difference of rates between privileged and unprivileged groups.

$$\mathsf{SFC}_{i\mathsf{TP}} = \begin{cases} 1 & \text{if } Y_i = \hat{Y}_i(\mathcal{P}) = 1 \text{ and } \hat{Y}_i(\mathcal{P}^*) = 0 \\ -1 & \text{if } \hat{Y}_i(\mathcal{P}) = 0 \text{ and } Y_i = \hat{Y}_i(\mathcal{P}^*) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{SFC}_{i\mathsf{FP}} = \begin{cases} 1 & \text{if } \hat{Y}_i(\mathcal{P}) = 1 \text{ and } Y_i = \hat{Y}_i(\mathcal{P}^*) = 0 \\ -1 & \text{if } Y_i = \hat{Y}_i(\mathcal{P}) = 0 \text{ and } \hat{Y}_i(\mathcal{P}^*) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{SFC}_{\mathsf{TP}} = \sum_{i=1}^{n} \mathsf{SFC}_{i\mathsf{TP}} \text{ , } \mathsf{SFC}_{\mathsf{FP}} = \sum_{i=1}^{n} \mathsf{SFC}_{i\mathsf{FP}}$$

$$\mathsf{SFR}_{\mathsf{AOD}}(u) = (1/2)\{\mathsf{SFC}_{\mathsf{TP}}(u)/n_{u,Y=1} + \mathsf{SFC}_{\mathsf{FP}}(u)/n_{u,Y=0}\}$$

$$\mathsf{SFR}_{\mathsf{AOD}}(p) = (1/2)\{\mathsf{SFC}_{\mathsf{TP}}(p)/n_{p,Y=1} + \mathsf{SFC}_{\mathsf{FP}}(p)/n_{p,Y=0}\}$$

$$\mathsf{SF}_{\mathsf{AOD}} = \mathsf{SFR}_{\mathsf{AOD}}(u) - \mathsf{SFR}_{\mathsf{AOD}}(p) \tag{5.5}$$

Finally, $\mathsf{SF}_{\mathsf{ERD}}$ is computed using the change of count in both false positives (FP) and false negatives (FN) as mentioned in the definition of ERD in Equation (5.1).

$$\mathsf{SFC}_{i\mathsf{FN}} = \begin{cases} 1 & \text{if } \hat{Y}_i(\mathcal{P}) = 0 \text{ and } Y_i = \hat{Y}_i(\mathcal{P}^*) = 1 \\ -1 & \text{if } Y_i = \hat{Y}_i(\mathcal{P}) = 1 \text{ and } \hat{Y}_i(\mathcal{P}^*) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{SFC}_{\mathsf{FN}} = \sum_{i=1}^{n} \mathsf{SFC}_{i\mathsf{FN}}$$

$$\mathsf{SFR}_{\mathsf{ERR}}(u) = \mathsf{SFC}_{\mathsf{FP}}(u)/n_{u,Y=0} + \mathsf{SFC}_{\mathsf{FN}}(u)/n_{u,Y=1}$$

$$\mathsf{SFR}_{\mathsf{ERR}}(p) = \mathsf{SFC}_{\mathsf{FP}}(p)/n_{p,Y=0} + \mathsf{SFC}_{\mathsf{FN}}(p)/n_{p,Y=1}$$

$$\mathsf{SF}_{\mathsf{ERR}} = \mathsf{SFR}_{\mathsf{ERR}}(u) - \mathsf{SFR}_{\mathsf{ERR}}(p) \tag{5.6}$$

Thus far, we have four fairness metrics ($\mathsf{SF}_{\mathsf{SPD}}$, $\mathsf{SF}_{\mathsf{EOD}}$, $\mathsf{SF}_{\mathsf{AOD}}$, and $\mathsf{SF}_{\mathsf{ERD}}$) to measure the fairness of the stage. In general, the rates computed by each metric (SFR) follow the same range of the original metrics

[-1, 1]. Therefore, the above metrics have a range [-2, 2]. Positive values indicate bias towards unprivileged group, negative values indicate bias towards privileged group, and values very close to 0 indicate fair preprocessing stage.

5.4 Evaluation

In this section, we describe the benchmark dataset and pipelines that we used for evaluation. Then we present the experiment design and results for answering the research questions.

5.4.1 Benchmark

We collected ML pipelines used in prior studies for fairness evaluation. First, Biswas and Rajan collected a benchmark of 40 ML models collected from Kaggle that operate on 5 different datasets e.g., *German Credit* [87], *Adult Census* [125], *Bank Marketing* [133], *Home Credit* [104] and *Titanic* [106]. However, the authors did not study the fairness at the component level, rather the ultimate fairness of the classifiers e.g., `RandomForest`, `DecisionTree`, etc. We revisited these Kaggle kernels and collected the preprocessing stages used in the pipelines. We noticed that *Home Credit* dataset [104] in this benchmark is not unified like the other datasets, distributed over multiples CSV files, and the models under this dataset do not operate on the same data files. Hence, these models (8 out of 40) are not suitable for comparing fairness of data preprocessing stages.

Second, we collected the pipelines provided by Yang et al. [207]. The authors released 3 pipelines on two different datasets - *Adult Census* and *Compas*. Third, Zelaya [211] studied the volatility of the preprocessing stages using two pipelines on a fairness dataset i.e., *German Credit*. We included these pipelines in our benchmark. Thus, we created a benchmark of 37 ML pipelines that operate on five datasets. The pipelines with the stages in each dataset category and their performances are shown in Table 5.1. Below we present a brief description of the datasets and associated tasks.

German Credit. The dataset contains 1000 data instances and 20 features of individuals who take credit from a bank [87]. The target is to classify whether the person has a good/bad credit risk.

Table 5.1: The preprocessing stages and performance measures (accuracy, f1 score) of the pipelines in the benchmark. * Fairness is measured with respect to a reference stage.

German	Stages	Acc	F1	Adult	Stages	Acc	F1
GC1	PCA, SB	0.64	0.76	AC1	SS, LE	0.85	0.66
GC2	SMOTE, SS	0.74	0.81	AC2	MV	0.85	0.68
GC3	PCA	0.73	0.83	AC3	Custom(f)	0.87	0.66
GC4	LE, SS	0.73	0.82	AC4	PCA, SS	0.85	0.66
GC5	SS	0.74	0.83	AC5	LE	0.87	0.71
GC6	PCA, SS, LE	0.73	0.83	AC6	Custom(f), Custom(c)	0.85	0.65
GC7	PCA, SB	0.66	0.77	AC7	PCA, SS, Custom(f)	0.78	0.51
GC8	SS	0.72	0.81	AC8	SS, Custom(f), Stratify	0.85	0.67
GC9	SMOTE	0.67	0.77	AC9	SS	0.81	0.61
GC10	Usamp	0.6	0.81	ACP10	Impute	0.81	0.62
Bank	**Stages**	**Acc**	**F1**	**Titanic**	**Stages**	**Acc**	**F1**
BM1	Custom, LE, SS	0.9	0.56	TT1	MV, Custom(f), Encode	0.77	0.83
BM2	LE	0.91	0.61	TT2	MV, Custom(f)	0.78	0.72
BM3	LE, SS, Custom(f)	0.9	0.48	TT3	Custom(f), Impute	0.8	0.72
BM4	SS	0.89	0.33	TT4	Custom(f), Impute, RFECV	0.81	0.73
BM5	SS	0.88	0.23	TT5	Custom(f)	0.83	0.76
BM6	FS, Stratify, SS	0.91	0.58	TT6	Custom(f)	0.82	0.74
BM7	FS	0.91	0.6	TT7	SS, LE, Custom(f)	0.82	0.77
BM8	Stratify	0.9	0.56	TT8	Custom(f)	0.83	0.76
Compas	**Stages**					**Acc**	**F1**
CP1	Filter, Impute1, Encode, Impute2, Kbins, Binarize					0.97	0.97
SB: SelectBest, SS: StandardScaler, LE*: LabelEncoder, Usamp: Undersampling, Custom(f/c): Custom feature engineering or cleaning, FS: Feature selection, MV: Missing value processing, RFECV: Feature selection method							

Adult Census. The dataset is extracted by Becker [125] from 1994 census of United States. It contains 32,561 data instances and 12 features including demographic data of individuals. The task is to predict whether the person earns over 50K in a year.

Bank Marketing. This dataset contains a bank's marketing campaign data of 41,188 individuals with 20 features [133]. The goal is to classify whether a client will subscribe to a term deposit.

Titanic. The dataset contains information about 891 passengers of Titanic [106]. The task is to predict the survival of the individuals on Titanic. The sensitive attribute of this dataset is *sex*.

Compas. The dataset contains data of 6,889 criminal defendants in Florida. Propublica used this dataset and showed that the recidivism prediction software used in US courts discriminates between White and non-White [10]. The task is to classify whether the defendants will re-offend where *race* is considered as the sensitive attribute.

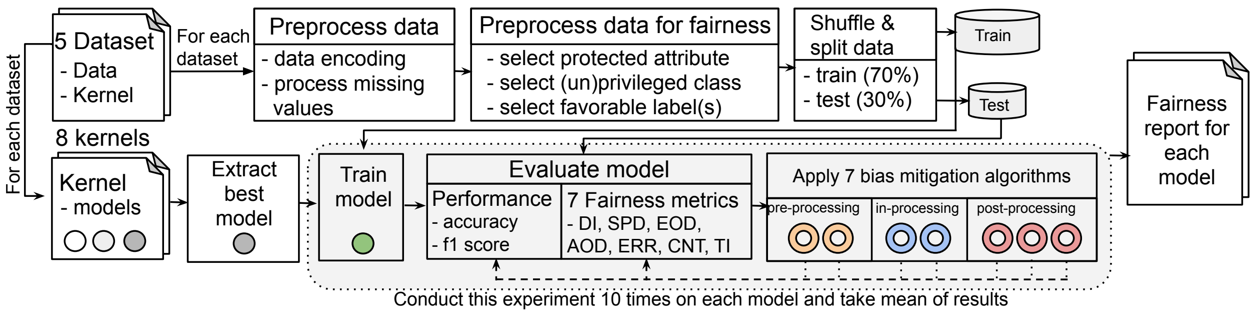

Figure 5.2: Experiment design to measure fairness of preprocessing stages in machine learning pipeline.

5.4.2 Experiment Design

Each pipeline in the benchmark consists of one or more preprocessing stages followed by the classifier. In this paper, our main goal is to evaluate the fairness of different preprocessing stages using the fairness metrics described in Section 5.3. The benchmark, code and results are released in the replication package [30].

The experiment design for evaluating the pipelines is shown in Figure 5.2. First, for each pipeline, we identified the preprocessing stages. For example, the pipeline in Section 5.2.1 contains six preprocessing stages. To evaluate the fairness of a stage S_k in a pipeline $\mathcal{P}$, we create an alternative pipeline $\mathcal{P}^*$ by removing the stage S_k. For stages that can not be removed, we replaced S_k with a reference stage S'_k. Among the preprocessing stages shown in Table 5.1, we found only the encoders can not be removed. We experimented with all the encoders in Scikit-Learn library [168], i.e., `OneHotEncoder`, `LabelEncoder`, `OrdinalEncoder`, and found that `OneHotEncoder` does not exhibit any bias. Therefore, we used `OneHotEncoder` as the reference stage for the encoders in our experiment.

Second, the original dataset is split into training (70%) and test set (30%). Then two copies of training data are used to train pipeline $\mathcal{P}$ and $\mathcal{P}^*$. After training the classifiers, two models predict the label for the same set of test data instances. Then, similar to the experimentation of [211], for each prediction label we compare the two predictions $\hat{Y}_i(\mathcal{P})$ and $\hat{Y}_i(\mathcal{P}^*)$ with the true prediction label Y_i. This comparison provides the necessary data to compute the four fairness metrics. Similar to [65, 27], for each stage in a pipeline, we run this experiment ten times, and then report the mean and standard deviation of the metrics, to avoid inconsistency of the randomness in the ML classifiers. Finally, we followed the ML best practices so that noise is not introduced evaluating the fairness of preprocessing stages. For example, while applying some transformation, lack of data isolation might introduce noise in the evaluation, e.g., when applying `PCA` on dataset, it is important to train the `PCA` only using the training data. If we use the whole dataset to train the `PCA` and transform data, then information from the test-set might leak. Third, since a stage operates on the data processed by the preceding stage(s), there are interdependencies between them. We always maintained the order of the stages while removing or replacing a stage (Section 5.5). To observe fairness of data transformers without interdependencies, we applied them on vanilla pipelines (Section 5.6).

5.5 Fairness of Preprocessing Stages

In this paper, we used a diverse set of metrics, to evaluate fairness of preprocessing stages. While developing an ML pipelines, if the developer has a comprehensive idea of the fairness of preprocessing stages, it would be convenient to build a fair pipeline. The evaluation has been done on 69 preprocessing stages in 37 ML pipelines from 5 dataset categories. For *Compas* dataset, we found one pipeline (Section 5.2.1). Five out of six stages in this pipeline exhibit no bias, which has been discussed later. For other 4 dataset categories, the evaluation has been shown in Figure 5.3. In this section, first, we discuss how we can interpret the metrics. Second, we answer the first research question and discuss the findings from our evaluation.

5.5.1 What Do the Metrics Imply?

We investigated the fairness of the preprocessing stages using four metrics: $\mathsf{SF}_{\mathsf{SPD}}$, $\mathsf{SF}_{\mathsf{EOD}}$, $\mathsf{SF}_{\mathsf{AOD}}$, and $\mathsf{SF}_{\mathsf{ERD}}$. These metrics measure the fairness of the stages by using the existing fairness criteria, e.g., $\mathsf{SF}_{\mathsf{SPD}}$ measures the fairness of a stage with respect to statistical parity difference (SPD) criteria. These fairness criteria evaluate algorithmic fairness of ML pipelines [64, 210, 36]. The unfairness characterized by these

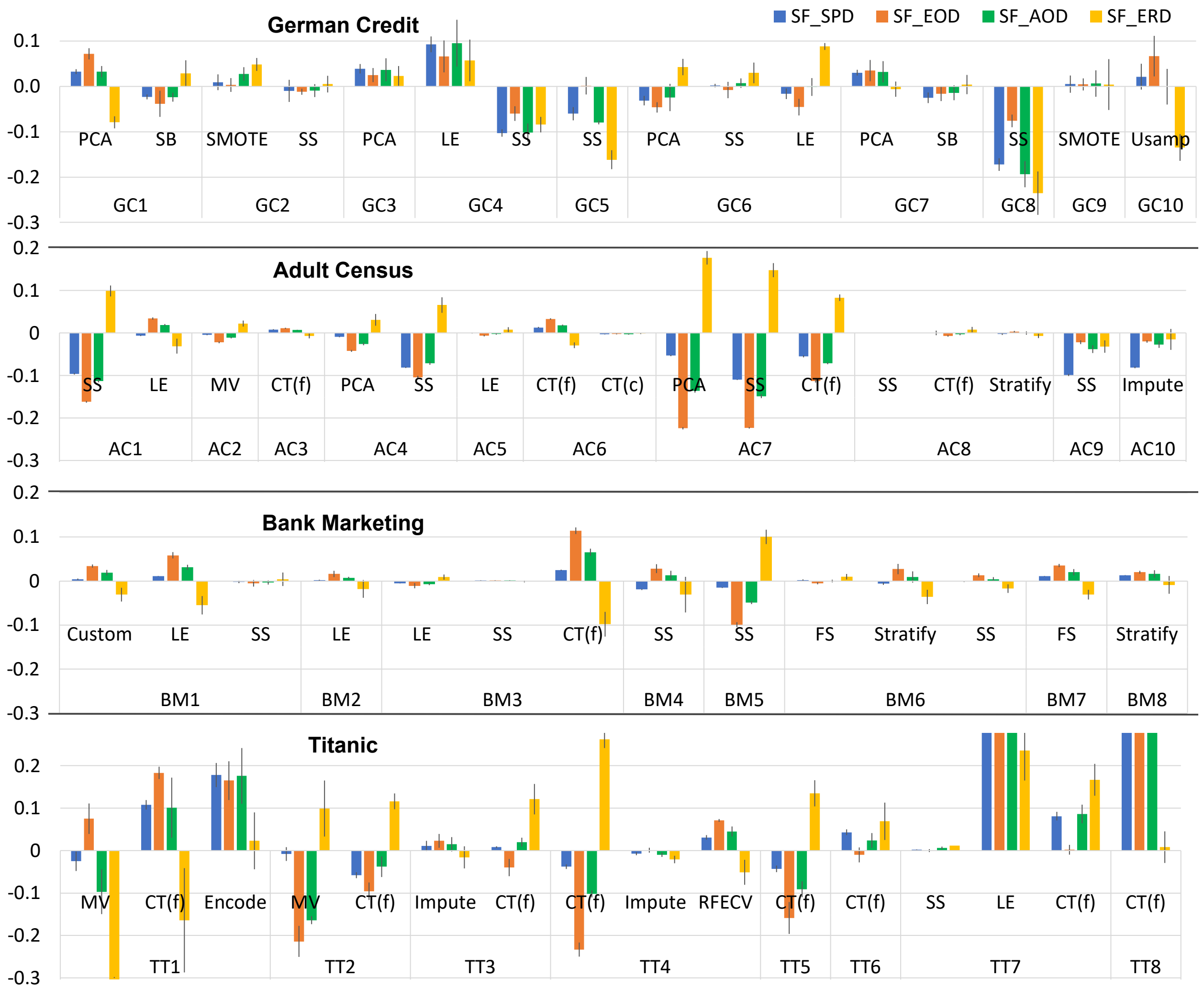

Figure 5.3: Fairness measures (Y-axis) of preprocessing stages in pipelines (X-axis). Grey lines above bars indicate standard error.

criteria is measured based on the prediction disparities, although the root cause can be the training data or the algorithm (e.g., data preprocessing, classifier) itself. Therefore, when an ML model is identified as unfair, it implies that in the given predictive scenario, the outcome is biased. Similarly, the metrics proposed in this paper measure algorithmic unfairness caused by a specific preprocessing stage with respect to its pipeline. For instance, in Figure 5.3, pipeline GC4 has two stages: `LabelEncoder` and `StandardScaler`. The fairness metrics suggest that `LabelEncoder` is biased towards unprivileged group (positive value), and `StandardScaler` is biased towards privileged group (negative value). The stages for which the measures are very close to zero, can be considered as fair preprocessing.

The metrics can provide different fairness signals for a certain stage. For example, in AC4, $\mathsf{SF}_{\mathsf{ERD}}$ shows positive fairness, whereas the other metrics suggest negative fairness for both the stages `PCA` and `StandardScaler`. This disparity occurs because different metrics accounts for different fairness criteria. In this case, $\mathsf{SF}_{\mathsf{ERD}}$, depends on the false positive and false negative rate difference. No other metric is concerned about the false negative rate difference, and hence $\mathsf{SF}_{\mathsf{ERD}}$ provides a different fairness signal than other metrics. In practice, appropriate fairness criteria can vary depending on the task, usage scenario, or involved stakeholders. Study suggests that developers need to be aware of different fairness indicators to build fairer pipelines [65]. Therefore, we defined and evaluated fairness of stages with respect to multiple metrics.

5.5.2 Fairness Analysis of Stages

The pipelines used both built-in algorithm imported from libraries i.e., *data transformers* [35], as well as *custom* preprocessing stages. The stages found in each pipeline are shown in Table 5.1, and the fairness measures of those stages are plotted in Figure 5.3. Although the unfairness exhibited by a stage is with respect to the pipeline, we found fairness patterns of some stages and investigated them further. In general, our findings show that the stages which change the underlying data distribution significantly, or modify minority data are responsible for increasing bias in the pipelines.

Finding 15: Data filtering and missing value removal change the data distribution and hence introduce bias in ML pipeline.

Most of the real-world datasets contain missing values (MV) for several reasons such as data creation errors, not-applicable (N/A) attributes, incomplete data collection, etc. In our benchmark, *Adult Census* and *Titanic* contain MV that required further processing in the pipeline. 7.4% rows in *Adult Census* and 20.2% rows in *Titanic* have at least one missing feature in the dataset. The pipelines either remove the rows with MV or apply certain imputation [169] technique that replaces the MV with mean, median or most frequently occurred values. Removal of rows with MV can significantly change the data distribution, which introduces bias in the pipeline. For example, both TT1 and TT2 removed data items with MV by applying `df.dropna()` method, which introduces bias in the prediction (Figure 5.3). Research has shown that MV are not uniformly distributed over all groups and data items from minority groups often contain more MV [54]. If those data items are entirely removed, the representation of minority groups in the dataset becomes scarce. On the other hand, TT3 applied mean-imputation and TT4 applied both median- and mode-imputation using `df.fillna()`, which exhibits fairness compared to data removal. While our findings suggest that removing data items with MV introduces bias, the most popular fairness tools AIF 360 [19], Aequitas [162], Themis-ML [15] ignore these data items and remove entire row/column. Our evaluation strategy confirms that the tools can integrate existing imputation methods [169] in the pipeline and allow users to choose appropriate ones. Additionally, more research is needed to understand and develop imputation techniques that are fairness aware.

Finding 16: New feature generation or feature transformation can have large impact on fairness.

We found that most of the feature engineering stages, especially the custom transformations exhibit bias in the pipeline. For example, the pipelines in *Titanic* dataset used custom feature engineering, since the dataset contains composite features which may provide additional information about the individuals. For instance, TT8 operates on the feature *name* to create a new feature *title* e.g., Mr, Mrs, Dr, etc. This transformation aims at better prediction of the survival of passengers by extracting the social status, but creates high bias between male and female.

In *Adult Census*, the feature *eduction* of individuals contain values such as *preschool*, *10th*, *1st-4th*, *prof-school*, etc., which have been replaced by broad categories such *Dropout*, *HighGrad*, *Masters* in the pipeline AC7. In addition, instead of using *age* as continuous value, the feature has been discretized into n number of bins. In both cases, the original data values have been modified, which has caused unfairness in the pipeline. Nevertheless, some pipelines (AC3, AC8) have custom feature transformations that are fair. Previous studies showed that certain features contribute more to the predictive quality of the model [71, 155]. Feature importance in prediction and corelation of features with the sensitive attribute also led to bias detection [41, 81] in ML models. However, does creating new features (by removing certain semantics) from a potentially biased feature increase the fairness, is an open question. Our method to quantify the fairness of such changes can guide further research in this direction.

Finding 17: Encoding techniques should be chosen cautiosly based on the classifier.

Two most used encoding techniques for converting categorical feature to numerical feature are `OneHotEncoder` and `LabelEncoder`. `OneHotEncoder` creates n new columns by replacing one column for each of the n categories. `LabelEncoder` does not increase the number of the columns, and gives each category an integer label between 0 and $(n-1)$. In our evaluation, we found that `LabelEncoder` introduces bias in *German Credit* and *Titanic* dataset but `OneHotEncoder` does not change fairness. Since `LabelEncoder` imposes a sequential order between the categories, it might create a linear relation with the target value, and hence have an impact on the classifier to change fairness. For example, pipelines TT7 creates a new feature called *Family* based on the surname of the person. This feature has a large number of unique categories (667 unique ones in 891 data instances). Therefore, the non-sparse representation in *LabelEncoder* adds additional weight to the feature, which is causing unfairness in TT7. Developers might avoid `OneHotEncoder` because it suffers from the curse of dimensionality and the ordinal relation of data is lost. In that case, developers should be aware of the fairness impact of the encoder. One solution might be using PCA for dimensionality reduction, which has been done in GC7.

Finding 18: The variability of fairness of preprocessing stages depend on the dataset size and overall prediction rate of the pipelines.

We have plotted the standard error of the metrics as error bars in Figure 5.3. Firstly, it shows that the metrics in *German Credit* and *Titanic* dataset are more unstable. The reason is that the size of these two datasets is less than the other three datasets. *German Credit* dataset has 1000 instances, and *Titanic* has 891 instances. *Adult Census* and *Bank Marketing* dataset have more than 30K instances. If the sample size is large, data distribution tends to be similar even after taking a random train-test split [65]. However, when the dataset size is smaller, the distribution is changed among different train-test splitting. Furthermore, we have found that $\mathsf{SF}_{\mathsf{ERD}}$ is more unstable than other metrics. $\mathsf{SF}_{\mathsf{ERD}}$ depends on the change of false positive and false negative rates. However, in most cases, the pipelines are optimized for accuracy and precision, since these are some best performing ones collected from Kaggle. Therefore, before deploying preprocessing stages, it would be desirable to test the stability of over multiples executions.

Finding 19: The unfairness of a preprocessing stage can be dominated by dataset or the classifier used in the pipeline.

For the *Compas* dataset, we evaluated the six stages shown in Section 5.2.1. All the stages exhibited data filtering show bias. The data filtering also showed bias close to zero (less than .005) with respect to all the metrics. Although Yang *et al.* [207] argued that this pipeline filters data in different proportions from *male* and *female* group, our evaluation confirms it does not cause unfairness. This pipeline has been used by Propublica [10] to show the bias in the prediction. Therefore, it is understandable that they did not employ any preprocessing that introduces bias in the pipelines. Other than that, almost all the preprocessing stages in *Bank Marketing* pipelines also exhibit very little unfairness, which suggest that the preprocessing on this dataset are fair in general.

A few stages show different behavior when they are used in composition with different classifiers. For example, `StandardScaler` has been applied on both GC6 and GC8. While GC6 employs a *RandomForest* classifier, GC8 uses *K-Neighbors* classifier. We have observed the opposite fairness measures for *StandardScaler* in these two pipelines. Therefore, fairness can be dominated by the underlying properties of data or the pipeline where it is applied. We have further investigated this phenomenon by applying transformers on different classifiers in the next section.

5.5.3 Fairness-Performance Tradeoff

In this section, we investigated the fairness-performance tradeoff for the preprocessing stages. The original performances of each pipeline have been reported in Table 5.1. To investigate fairness of a stage, we created pipeline $\mathcal{P}^*$ by removing the stage from original pipeline $\mathcal{P}$ (Section 5.2). To understand fairness-performance tradeoff, we evaluated performance (both accuracy and f1 score) of $\mathcal{P}^*$ and $\mathcal{P}$ in the same experimental setup. Then we computed the performance difference to observe the impact of the stage

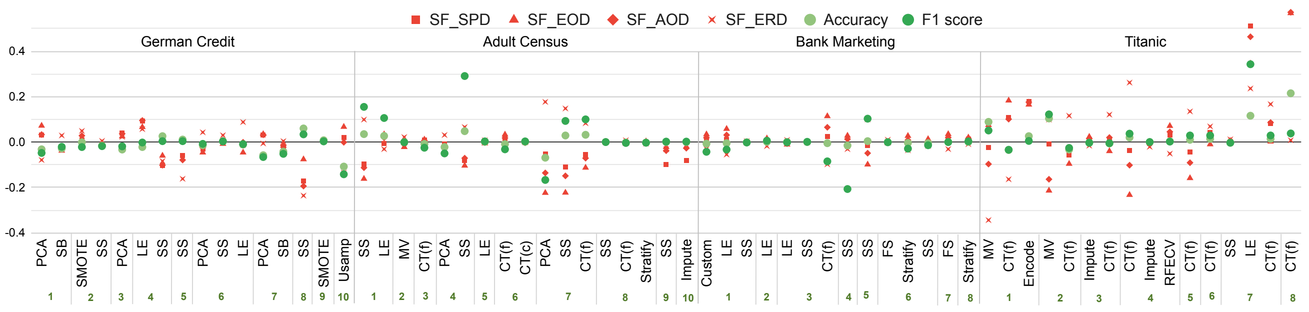

Figure 5.4: Performance changes (green) are plotted with fairness (red) of the preprocessing stages. A positive or negative performance change indicates performance increase or decrease respectively, after applying the stage.

on performance. For example, $Acc(\mathcal{P}) - Acc(\mathcal{P}^*)$ gives the accuracy increase (or decrease, if negative) after applying a stage. We plotted the performance impacts of the stages with their fairness measures in Figure 5.4.

First, many preprocessing stages have negligible performance impact. In Figure 5.4, 19 out 63 stages exhibits accuracy and f1 score change in the range [-0.005, 0.005], which indicates performance change $\leq$ 0.05%. We found that in all of these cases, except AC9 and AC10, the preprocessing stages are fair with a very small degree of bias. Second, tradeoff between performance and fairness is observed for the stages which improve performance. 17 stages improve accuracy or f1 score more than 0.05%, which further exhibits moderate to high degree of bias. Overall, the most biased stages - TT7(LE), TT8(CT), TT4(CT), TT1(MV), GC8(SS), are improving performance. This stage-specific tradeoff is aligned with the overall performance-fairness tradeoff discussed in prior work [27, 65, 42], which can be compared quantitatively by the work of Hort et al. [90]. Third, we found that some stages decrease the performance, either accuracy or f1 score. Surprisingly, most of these stages also exhibit high degree of bias. For instance, the most performance-decreasing stages - BM4(SS), AC7(PCA), GC10 (Undersampling), are showing more bias. Our fairness evaluation would facilitate developers to identify and remove such stages in the pipeline.

5.6 Fair Data Transformers

In Section 5.5, we found that many data preprocessing stages are biased. Many bias mitigation techniques applied in preprocessing stage have been shown successful [64, 113]. If we process data with appropriate transformer, then it might be possible to avoid bias and mitigate inherent bias in data or classifier.

Table 5.2: Transformers collected from ML pipelines and libraries

Categories	Stages	Transformers
MV processing	Imputation	SimpleImputer, IterativeImputer
Categorical encoding	Encoder	Binarizer, KBinsDiscretizer, LabelBinarizer, LabelEncoder, OneHotEncoder
Standardization	Scaling	StandardScaler, MaxAbsScaler, MinMaxScaler, RobustScaler
	Normalization	l1-normalizer, l2-normalizer
Feature engineering	Non-linear transformation	QuantileTransformer, PowerTransformer
	Polynomial feature generation	PCA, SparsePCA, MiniBatchSparsePCA, KernelPCA
	Feature selection	SelectKBest, SelectFpr, SelectPercentile
Sampling	Oversampling	SMOTE
	Undersampling	AllKNN
	Stratification	Random, Stratified

Even if a data transformer is biased towards a specific group, it could be useful to mitigate bias if original data or model exhibits bias towards the opposite group. To that end, we want to investigate the fairness pattern of the data transformers. However, in our evaluation (Figure 5.3), some transformers have been used only in specific situations e.g., `SMOTE` has been only applied on *German Credit* dataset. What is the fairness of this transformer when used on other datasets and classifier? In this section, we setup experiments to evaluate the fairness of commonly used data transformers on different datasets and classifiers.

First, we collected the classifiers used in each dataset category from the benchmark. Then, for each dataset, we created a set of vanilla pipelines. A vanilla pipeline is a classification pipeline which contain only one classifier. Second, we found a few categories of preprocessing stages from our benchmark shown in Table 5.2. For each transformer used in each stage, we collected the alternative transformers from corresponding library. For example, in our benchmark, `StandardScaler` from Scikit-Learn library has been used for scaling data distribution in many pipelines. We collected other standardizing algorithms available in Scikit-Learn. We found that besides `StandardScaler`, Scikit-Learn also provides `MaxAbsScaler`, `MinMaxScaler`, and `Normalizer` standardize data [168]. Similarly, a data oversampling technique `SMOTE` has been used in the benchmark, we collected another undersampling technique `ALLKNN` and a combination of over- and undersampling sampling technique `SMOTENN` from

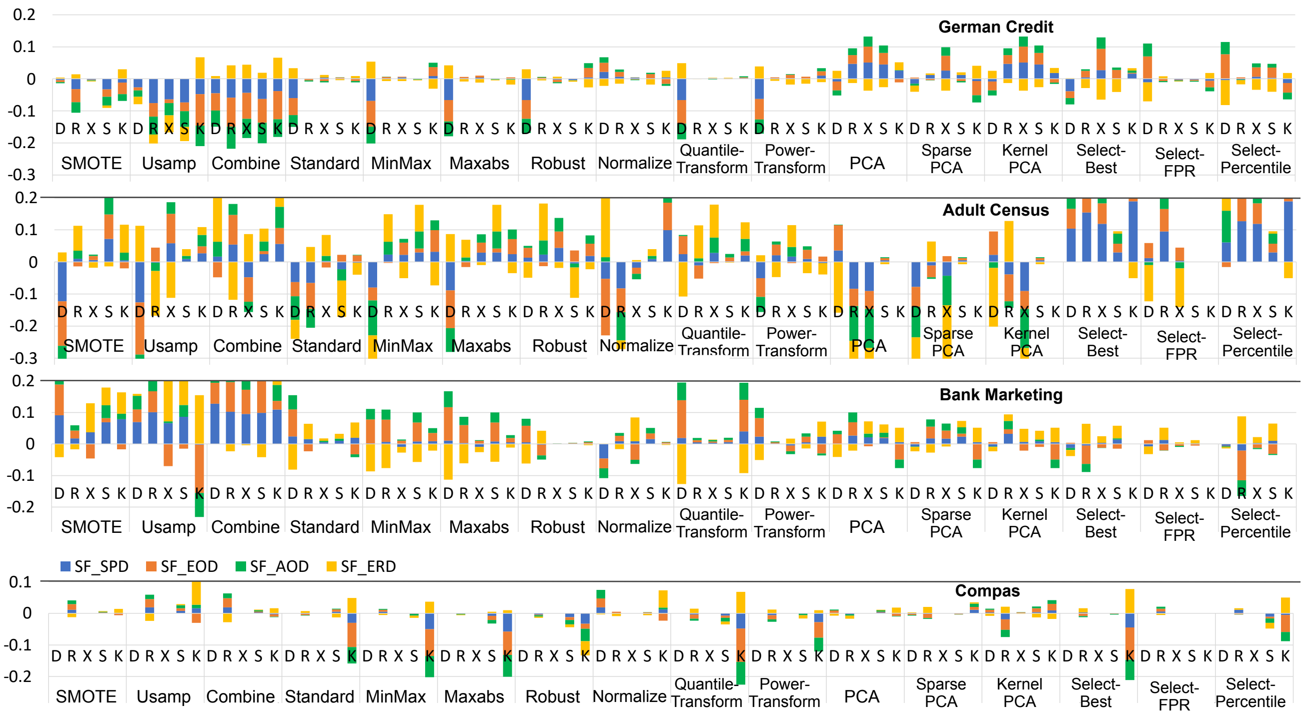

Figure 5.5: Fairness impact of data transformers on ML classifiers, D: `DecisionTree`, R: `RandomForest`, X: `XGBoost`, S: `SupportVector`, K: `KNeighbors`

IMBLearn library [128]. Third, in each of the vanilla pipelines, we applied the transformers and evaluated fairness using the method used in Section 5.4.2 with respect to four metrics. We found that pipelines under *Titanic* uses custom transformation, and most of the built-in transformers are not appropriate for this dataset. So, to be able to make the comparison consistent, we conducted this evaluation on four datasets: *German Credit*, *Adult Census*, *Bank Marketing*, *Compas*. Finally, we did not use transformers for imputation and encoder stages. Encoding transformers (LabelEncoder, OneHotEncoder), have been applied on most of the pipelines and their behavior has been understood. The fairness measures of each transformer on different classifiers have been plotted in Figure 5.5.

Fairness among the datasets follows a similar pattern. This further confirms that the unfairness is rooted in data. The *Compas* dataset shows the least bias. Although racial discrimination has been reported for this dataset [10], this is a more curated dataset than the other three. By looking at the overall trend of fairness, we observe that sampling techniques have the most biased impact on prediction. Other than that feature selection transformers have more impact than other ones.

Finding 20: Among all the transformers, applying sampling technique exhibits most unfairness.

Sampling techniques are often used in ML tasks when dataset is class-imbalanced. Unlike the other transformers, sampling techniques make horizontal transformation to the training data. The oversampling technique `SMOTE` creates new data instances for the minority class by choosing the nearest data points in the feature space. Undersampling techniques balance dataset by removing data items from majority class. Although balancing dataset has been shown to increase fairness [54], our evaluation suggest that in three out of four datasets, it increases bias.

From Figure 5.5, we can see that sampling techniques exhibit the most unfairness. In *German Credit* dataset, different classifier reacts differently when sampling is done. `DecisionTree` classifier exhibits most unfairness for both oversampling and undersampling towards privileged group i.e., *male*. Interestingly, the combination of over- and undersampling also fails to show fairness. Furthermore, both *German Credit* and *Bank Marketing* pipelines exhibit bias towards unprivileged group, which might be desired when compared to bias towards privileged.

Finding 21: Selecting subset of features often increase unfairness.

Selecting the best performing feature can give performance improvement of the pipeline. However, unfairness can be encoded in specific features [81]. While selecting best features, some features which encodes unfairness, can dominate the outcome. Thus, many classifiers in *German Credit*, *Adult Census*, and *Bank Marketing* show unfairness because of reduced number of features, which has been also observed by Zhang and Harman [215]. Surprisingly, `SelectFpr` exhibited very little or no bias compared to the other feature selection methods. A detailed investigation suggests that `SelectBest` and `SelectPercentile` select only the k most contributing features. However, `SelectFpr` performs false positive rate test on each feature, and if it falls below a threshold, the feature is removed [166]. Therefore, it does not apply harsh pruning, which contributes to the fairness of the prediction.

Finding 22: In most of the pipelines, feature standardization and non-linear transformers are fair transformers.

These transformers modify the mean and variance of the data by applying linear or non-linear transformation. However, they do not change the feature importance on the classifiers. Therefore, in most of the cases, these transformers (especially, `StandardScaler` and `RobustScaler`) are fair. Some classifiers show bias after applying these transformers such as, KNC in *Compas*. The unfairness exhibited by those pipelines are introduced by the classifiers, since these classifiers show similar bias pattern for other transformers as well. The scalers can impact the fairness significantly if there are many outliers in data. That is why we see more bias for the scalers in *German Credit* dataset. Therefore, although standardizing transformers are fair in general, they can be biased in composition with specific classifier or data property.

5.7 Fairness Composition of Stages

From our evaluation, we found that many data transformers have fairness impact on ML pipeline. In this section, we compare the *local fairness* (fairness measures of preprocessing stages) with the *global fairness* (fairness measures of whole pipeline). First, we answer whether the local fairness composes in the global fairness. Second, we investigate if we can leverage the composition to mitigate bias by choosing appropriate transformers.

5.7.1 Composition of Local and Global Fairness

We evaluated the global fairness of *Adult Census* pipelines (Table 5.1) using the four existing metrics from Equation (5.1). We calculated the fairness difference of these pipelines before and after applying the preprocessing stages. Additionally, we have evaluated the stage-specific fairness metrics. Both the local fairness and difference in global fairness of those pipelines have been plotted in Figure 5.6.

We can see that local and global fairness follow the same trend in most of the pipelines. This confirms that local fairness is directly contributing to the global fairness. However, the global fairness is computed based on the overall change in the prediction, whereas the local fairness considers the predictions for only

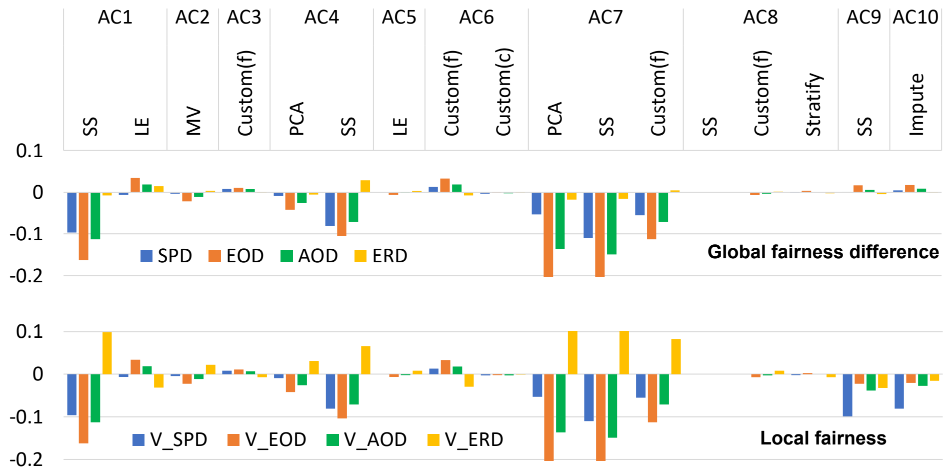

Figure 5.6: Comparison of global fairness change and local fairness for *Adult Census* dataset pipelines

those data instances which have been altered after applying a transformer Equation (5.3). For example, in Figure 5.6, for some pipelines (e.g., AC9, AC10), global and local fairness exhibit different trends. In these cases, the overall classification rate difference is not similar to the rate difference of altered labels. This means that the stages changed the labels such that it shows bias towards privileged. But when those changes in the labels are considered in addition to all the labels (global fairness), the bias difference could not capture the actual impact of that stage. We have verified this observation by manually inspecting the altered prediction labels. Thus, we can conclude that the local fairness composes to the global fairness. Specifically, if a preprocessing stage shows bias for privileged group, it pulls the global fairness towards the fairness direction of privileged group. However, only observing the global fairness difference, we can not measure the fairness of a given stage or transformer.

5.7.2 Bias Mitigation Using Appropriate Transformers

For a given transformer in an ML pipeline, a *downstream* transformer operates on data already processed by the given transformer and an upstream transformer is applied before the given one. Since the fairness of a preprocessing stage composes to the global fairness, can we choose a downstream transformer to mitigate bias in ML pipeline? In this section, we empirically show that the global unfairness can be mitigated by choosing the appropriate downstream transformer.

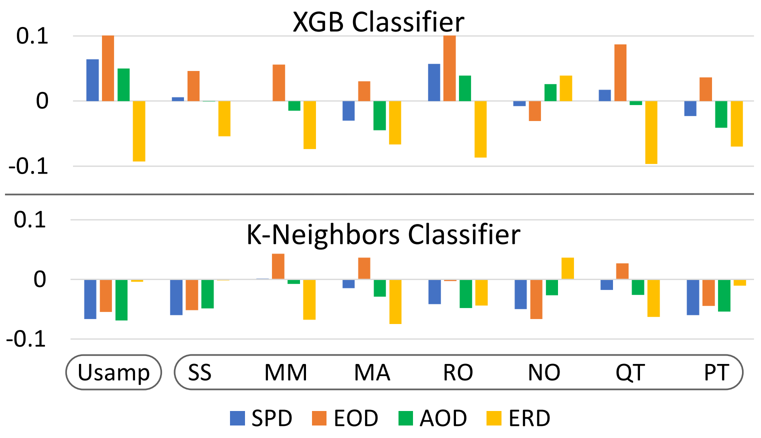

Figure 5.7: Global fairness after applying the upstream transformer (left), and after applying both the upstream and one downstream transformer (right). Usamp: undersampling.

Consider the use-case of classification task on *German Credit* dataset with different classifiers similar to Figure 5.5. Suppose, the original pipeline is constructed using undersampling technique. Since this pipeline exhibits bias, as shown in Figure 5.5, can we choose a downstream data standardizing transformer that mitigates that bias? In this use case, undersampling is the upstream transformer, and any standardizing transformer is the downstream transformer.

We showed the evaluation for XGB classifier and KNC classifier, since these two exhibits most bias when the upstream transformer was applied in Section 5.6. We plotted the global fairness after applying only the upstream transformer in the left of Figure 5.7. We also reported the local fairness of the standardizing transformers in Table 5.3. Now, since undersampling method exhibits bias towards privileged group for XGB, we look for the transformer that is biased towards privileged group. In Figure 5.7, among other transformers, `Normalizer` is the most successful to mitigate bias of the upstream transformer. Similarly, for KNC, the upstream operator exhibits bias towards privileged group. From Table 5.3, we can see that `MinMaxScaler` is the most biased transformer towards the opposite direction. As a result, applying `MinMaxScaler` mitigates bias the most. Note that the other downstream transformers also follow the fairness composition with its upstream transformer. Therefore, by measuring fairness of the preprocessing stages, developers would be able to instrument the biased transformers and build fair ML pipelines.

Table 5.3: Local fairness of stages as downstream transformer. SS: StandardScaler, MM: MinMaxScaler, MA: MaxAbsScaler, RO: RobustScaler, NO: Normalizer, QT: QuantileTransformer, PT: PowerTransformer.

Model	Stage	SF_SPD	SF_EOD	SF_AOD	SF_ERD
XGB	SS	0.001	0.021	-0.016	-0.073
	MM	-0.006	0.031	-0.029	-0.12
	MA	-0.036	0.005	-0.059	-0.128
	RO	0.051	0.082	0.025	-0.113
	NO	-0.014	-0.056	0.012	0.135
	QT	0.011	0.062	-0.02	-0.165
	PT	-0.029	0.011	-0.055	-0.132
KNC	SS	0.035	0.025	0.055	0.059
	MM	0.095	0.12	0.096	-0.05
	MA	0.079	0.114	0.075	-0.078
	RO	0.052	0.074	0.056	-0.036
	NO	0.045	0.010	0.077	0.134
	QT	0.077	0.104	0.078	-0.052
	PT	0.035	0.032	0.050	0.036

5.8 Discussion

We took the first step to understand the fairness of components in ML pipelines. Our method helps to provide causality in software and reason about behavior of components based on the impact on outcome. This method can be extended further to evaluate the fairness of other software modules [140] in ML pipeline and localize faults [201]. Moreover, we found most of the stages exhibited bias, to a low or higher degree. The fairness measures of different components can be leveraged towards fairness-aware pipeline optimization to satisfy fairness constraints. For example, US Equal Employment Commission suggests selection-rate difference between groups less than 20% [192]. Also, pipeline optimization techniques, e.g., TPOT [138], Lara [126] can be potentially utilized for pipeline optimization.

Furthermore, research has been conducted to understand the impact of preprocessing stages with respect to performance improvement [49, 193, 43]. This paper will open research directions to develop preprocessing techniques that improve performance by keeping the fairness intact. We also reported a number of fairness patterns of preprocessing stages that inducing bias in the pipeline such as missing value processing, custom feature generation, feature selection. Moreover, instrumentation of the stages can mitigate the inherent bias of the classifiers. It shows opportunities to build automated tools for identifying fairness bugs in AI systems and recommending fixes [95, 94]. Finally, current fairness tools (e.g., AIF 360 [19], Aequitas [162]) can be augmented by incorporating data preprocessing stages into the pipelines and letting users have control over the data transformers and observe or mitigate bias. Similarly, the libraries can provide API support to monitor fairness of the transformers.

5.8.1 Threats to Validity

Internal validity refers to whether the fairness measures used in this paper actually captures the fairness of preprocessing stages. To mitigate this threat, we used existing concepts and metrics to build new set of metrics. Causality in software [144, 145] has been well-studied, and causal reasoning in fairness has also been popular [127, 215, 163, 160], since it can provide explanation with respect to change in the outcome. Besides, this method do not require an oracle because the prediction equivalences provide necessary information to measure the impact of the intervention [66]. Furthermore, in Section 5.7, we conducted experiments on local and global fairness to show how new metrics composes in the pipeline.

External validity is concerned about the extent the findings of this study can be generalized. To alleviate this threat, we conducted experiments on a large number of pipeline variations. We collected the pipelines from three different sources. Moreover, we collected alternative transformers from the ML libraries for comparative analysis. Finally, for the same dataset categories, we used multiple classifiers and fairness metrics so that the findings are persistent.

5.9 Related Works

Dwork and Ilvento argued that fairness is dynamic in a multi-component environment [57]. They showed that when multiple classifiers work in composition, even if the classifiers are fair in isolation, the overall system is not necessarily fair. Bower *et al.* discussed fairness in ML *pipeline*, where they considered *pipeline* as sequence of multiple classification tasks [32]. They also showed that when decisions of fair components are compounded, the final decision might not be fair. For example, while interviewing candidates in two stages, fair decision in each stage may not guarantee a fair selection. D'Amour *et al.* studied the dynamics of fairness in multi-classification environnement using simulation [51]. In these research, fairness composition is shown over multiple tasks and the authors did not consider fairness of components in single ML pipeline. We position our paper here to study the impact of preprocessing stages in ML pipeline and evaluate the fairness composition.

5.10 Summary

Data preprocessing techniques are used in most of the machine learning pipelines in composition with the classifier. Studies showed that fairness of machine learning predictions depends largely on the data. In this paper, we investigated how the data preprocessing stages affect fairness of classification tasks. We proposed the causal method and leveraged existing metrics to measure the fairness of data preprocessing stages. The results showed that many stages induce bias in the prediction. By observing fairness of these data transformers, fairer ML pipelines can be built. In addition, we showed that existing bias can be mitigated by selecting appropriate transformers. We released the pipeline benchmark, code, and results to make our techniques available for further usages. Future research can be conducted towards developing automated tools to detect bias in ML pipeline stages and instrument that accordingly.

CHAPTER 6. FAIRIFY: FAIRNESS VERIFICATION OF NEURAL NETWORKS

Fairness of machine learning (ML) software has become a major concern in the recent past. Providing formal guarantees in ML models is challenging because of the complex decision-making process of the models. In this chapter, we propose an approach to verify individual fairness property in neural networks (NN) models. Individual fairness ensures that any two similar individuals should be predicted similarly irrespective of their protected attributes e.g., race, sex, age. Although individual fairness property has been widely used in testing literature, verifying individual fairness is hard because of the global nature of the property and presence of non-linear computation nodes in NN. We propose *Fairify* that makes the individual fairness verification tractable for the NN in production. Fairify leverages white-box access and neural pruning to provide certification or counterexample. The key idea is that many neurons in the NN always remain inactive for certain smaller part of the input domain. So, Fairify applies input partitioning and then prunes the NN for each partition to make them amenable to verification. Fairify uses sound neural pruning based on interval arithmetic and individual verification. In addition, Fairify analyzes activation heuristics of neurons to further prune NN. We propose the first SMT-based fairness verification that can answer targeted fairness queries with relaxations and provide counterexamples. Furthermore, Fairify is configurable based on the input domain and size of the NN. We evaluated Fairify on 25 real-world neural networks collected from four different sources, and demonstrated the effectiveness, scalability and performance over baseline and closely related work.

6.1 Introduction

Artificial intelligence (AI) based software are increasingly being used in critical decision making such as criminal sentencing, hiring employees, approving loans, etc. Algorithmic fairness of these software raised significant concern in the recent past [5, 27, 29, 40, 90, 39, 216, 217]. Several studies have been conducted to measure and mitigate algorithmic fairness in software [36, 48, 64, 84, 56, 181, 210, 213, 148, 212]. However, providing formal guarantees of fairness is difficult given the complex decision making process of the machine learning (ML) algorithms and the specification of the fairness properties in real-world [34, 6, 97, 17]. Our goal in this chapter is to address the fairness verification problem of neural network (NN) models.

Albarghouthi et al. and Bastani et al. proposed probabilistic techniques to verify *group fairness* [6, 17]. Group fairness property ensures that the protected groups (e.g., male-vs-female, young-vs-old, etc.) get similar treatment in the prediction. On the other hand, *individual fairness* states that any two similar individuals who differ only in their protected attribute get similar treatment [66, 97]. Galhotra et al. argued that group fairness property might not detect bias in scenarios when same amount of discrimination is made for any two groups [66], which led to the usage of individual fairness property in many recent works [66, 217, 5, 190, 218].

We propose *Fairify*, the first technique to verify *individual fairness* of NN models in production. Both abstract interpretation [151, 191, 131, 72, 178] and SMT based techniques [92, 116, 118] have shown success in verifying properties of NN. Recent work proposed methods to guarantee individually fair learning by enforcing it during model training [159, 209]. Urban et al. proposed abstract interpretation based *dependency fairness* certification for NN [191]. John et al. proposed individual fairness verification for two classes of ML classifiers: 1) linear classifier e.g., logistic regression, and 2) kernelized classifier e.g., support vector machine [97]. Our work differs from [159, 209] since we consider verifying already trained model statically, similar to [191]. However, unlike [191] we verify *individual fairness* of NN using SMT based technique. Thus, we can provide *counterexample* when there is a fairness violation, and we can verify various *relaxation* of fairness queries [97], which was not tackled in [191]. Furthermore, unlike [97] we considered verifying NN, which require different approach than ML classifiers.

Verifying a property in NN is challenging mainly because of the presence of non-linear computation nodes i.e., activation functions [116, 191]. With the size of the NN, the verification task becomes harder and often untractable [183, 199]. Many studies have been conducted to verify NN for different local robustness properties [72, 92, 61]. However, individual fairness property requires global checking which makes the verification task even harder and existing local property verifiers can not be used [78, 117]. To that end, we propose a novel technique that can verify fairness, i.e., provide SAT with counterexample or show UNSAT in

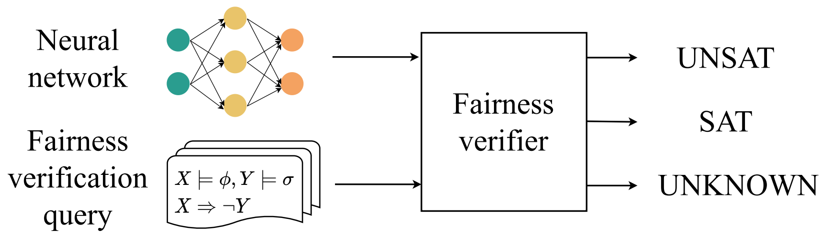

Figure 6.1: Fairness verification problem in neural networks

a tractable time and available computational resource. In the verification problem, the verifier takes a trained NN and the verification query as input. If the verifier can verify within the given timeout period, the output should be SAT which means there is a violation of the property, or UNSAT which means the property holds. When the verifier is unable to show any proof within the given timeout, the result is UNKNOWN. Our evaluation shows that using only the SMT solver, we cannot verify the fairness property of the NN models in days; however, Fairify can verify most of the verification queries within an hour. Fairify makes the verification task tractable and scalable by combining input partitioning and pruning approach.

Our key insight is that the activation patterns of the NN used in practice are sparse when we consider only a smaller part of the input region. Therefore, we perform **input partitioning** to a certain point and divide the problem into multiple sub-problems which result into split-queries. To show the problem as UNSAT, all the split-queries have to be UNSAT. On the other hand, if any of the split-queries gives SAT, the whole problem becomes SAT. Because of this construction of the problem, Fairify is able to provide certification or counterexample for each partition which has further value in fairness defect localization.

The input partitioning allowed us to perform static analysis on the network for the given partition and identify inactive neurons. We applied interval arithmetic to compute bounds of the neurons. In addition, we run individual verification query on each neuron to identify the ones that are always inactive. Since the inactive neurons do not impact the decision, removing those neurons gives a pruned version of the network that can be verified for the given partition. This **pruning technique** is lightweight, sound and achieves high pruning ratio, which makes the verification task tractable.

We further noticed that many neurons can not be removed based on the sound pruning but they remain inactive *almost* always. We conduct lightweight simulation to profile the network and find such candidate neurons. Then we leverage the layer-wise **heuristics** that suggest inactive nodes for the given partition. This approach is applied on top of sound pruning, if necessary given the time budget. Although this pruning method is based on heuristics, our evaluation shows that a conservative approach provides much improvement in the verification with negligible loss of accuracy. Another novelty in this idea is that the developer can choose to deploy the pruned (but verified) version of the NN. Thus, the pruned NN might have little accuracy-loss but would guarantee fairness.

After partitioning and reducing the complexity of the problem, we leverage a constraint solver to verify the split-queries on the pruned version of the networks. Then we accumulate the results for each partition to provide verification for the original query. We evaluated Fairify on 25 different NN models collected from four different sources. We collected appropriate real-world NNs from Kaggle which are built for three popular fairness-critical tasks and took the NNs used in three prior works in the area [217, 190, 191]. Our results show significant improvement over the baseline with respect to both utility, scalability, and performance. The main contributions of our work are as follows:

1. Fairify is the first to solve the individual fairness verification problem for NN using SMT based technique.

2. We showed how individual fairness and its relaxation can be encoded into verification queries. Furthermore, we proposed two novel NN pruning methods designed for fairness verification.

3. Fairify provides certification or counterexample for each input partition. Because of our construction of the problem, it is possible to provide targeted certification as well as suggest defect localization for fairness.

4. We implemented Fairify using Python and openly available constraint solver Z3. We also created a benchmark of NN models for fairness verification. The code, models, benchmark datasets, and results are available in the self-contained anonymous GitHub repository[1] that can be leveraged in future research.

The rest of the chapter is organized as follows: Section 6.2 describes the NN in consideration and Section 6.3 introduces fairness verification problem in NN. Section 6.4 provides detailed description of the

[1]https://github.com/anonymous-authorss/Fairify

approach and algorithms to verify fairness. Section 6.5 describes the results of our evaluation and answers the research questions. Finally, Section 6.6 describes the related work, and Section 6.7 concludes.

6.2 Preliminaries

Neural Networks (NN). We consider NN as a directed acyclic graph (DAG), where the nodes hold numeric values that are computed using some functions, and edges are the data-flow relations. The nodes are grouped into layers: one input layer, one or more hidden layers, and one output layer.

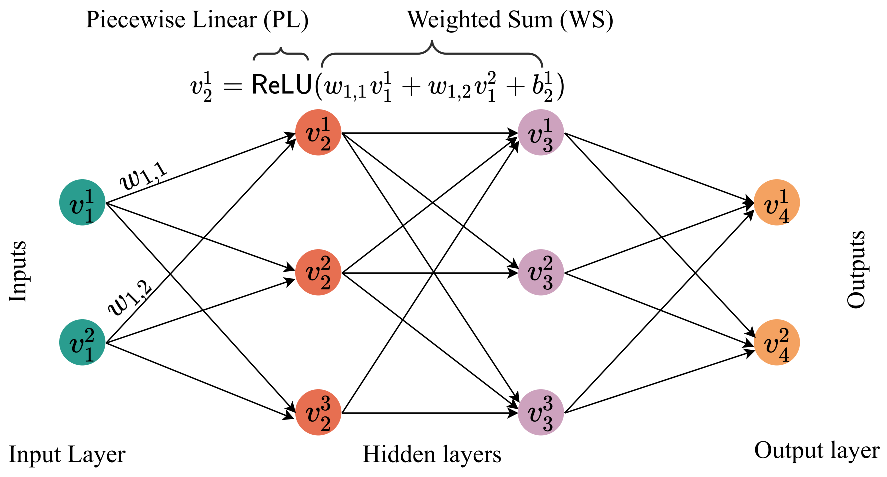

Figure 6.2: A fully-connected NN with ReLU activation

More formally, a NN model $M : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a DAG with k layers: $L_1, L_2, \ldots, L_k$ where L_1 is the input layer and L_k is the output layer. The size of each layer L_i is denoted by s_i and layer L_i has the nodes $v_i^1, v_i^2, \ldots, v_i^{s_i}$. Therefore, $s_1 = n$ (number of inputs) and $s_k = m$ (number of output classes). In this paper, we consider the fully connected networks where the functions to calculate the value in a node is given by $v_i^j = \mathsf{NL}(\Sigma_t w_{i-1,t} \cdot v_{i-1}^t + b_i^j)$. Here, node v_i^j is computed as weighted sum of each neuron v_{i-1}^t from previous layer, and the bias b_i^j assigned for the node. The weights $w_{l,t}$ and biases are constant real values in a trained NN, which are learned in the training phase. After computing the weighted sum, a non-linear activation function NL is applied to compute the final value of each node v_i^j. In practice, the ReLU activation function is widely used as it demonstrates good performance, and following the prior works in the area, we also considered ReLU based NN in this paper [206, 116, 143, 191]. ReLU is given by Eq. Equation (6.1) which is a piecewise-linear function.

$$\mathsf{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \text{ ; inactive neuron} \\ x & \text{if } x > 0 \text{ ; active neuron} \end{cases} \tag{6.1}$$

The neurons in the input layer accept data input values, which are passed to the following layers through the edges. Each neuron in the hidden and output layer computes its value by applying the weighted sum (WS) function and then the piecewise linear (PL) function. Each edge in the DAG is associated with a weight. Then the PL function (ReLU for all hidden neurons) transforms the WS value into the final value of the neuron. The output neurons may have different PL functions (e.g., Sigmoid, Softmax) that computes the predictive classes in classification problem.

6.3 Fairness Verification

In this section, we define the individual fairness verification problem and compare with other closely related NN verification problem.

NN Verification. A property $\phi(M)$ of the NN model M defines a set of input constraints as precondition ϕ_x and a set of output constraints as postcondition ϕ_y. Several safety properties of NN have been investigated in the literature [134, 75]. One of the most studied property is adversarial robustness [92, 91]. Given a model M and input x_0, adversarial robustness ensures that a minimum perturbation to x_0 does not change the label predicted by the model, i.e., $\forall x'.\ ||x_0 - x'|| \leq \delta \Rightarrow M(x_0) = M(x')$. The property ensures *local* safety guarantee around the neighborhood of x_0 with a distance δ. Different approaches have been proposed to guarantee safe region around the given input [37, 75] or generate targeted attacks that violate the property [184, 134, 75]. Unlike the robustness property, individual fairness requires *global* safety guarantee.

6.3.1 Individual Fairness

Group fairness property considers average-case fairness, where some notion of parity is maintained between protected groups. Individual fairness considers worst-case fairness, i.e., all similar input pairs get similar outcome [66, 97]. Suppose, $\mathcal{D} = \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ is a dataset containing n data instances, where each instance $\mathsf{x} = (x_1, \ldots, x_t)$ is a tuple with t attribute values. The set of attributes is denoted by $A = \{A_1, \ldots, A_t\}$ with its domain I, where $A_i = \{a \,|\, a \in I_i\}$. The set of protected attributes is denoted by P where $P \subset A$. So, individual fairness is defined as follows, which is widely used in the literature [217, 5, 190, 218].

Definition 1 ***(Individual Fairness)*** *The model M is individually fair if there is no such pair of data instances (x, x') in the input domain such that:* ❶ $x_i = x'_i$ *,* $\forall i \in A \setminus P$*,* ❷ $x_j \neq x'_j$ *,* $\exists j \in P$*, and* ❸ $M(x) \neq M(x')$*.*

Intuitively, individual fairness ensures that any two individuals who have same attribute values except the protected attributes, get the same prediction. If there exists such pair, the property is violated and (x, x') is considered as an individual bias instance. For example, suppose while predicting income of individuals ($>$ \$50K or not), *race* is considered as the protected attribute. Definition 1 suggests that if two individuals (x, x') with different *race* but exact same non-protected attributes, e.g., *occupation*, *age*, *marital-status*, are predicted to the different class, then the model is unfair.

The above definition is similar to the global robustness property introduced by [116, 117]. The global property is significantly harder to check since both the inputs x and x' can obtain any value within the domain, whereas in local robustness an input is fixed [78, 117]. To check such global property, two copies of the same NN are encoded in the postcondition to check $M(x) \neq M(x')$. Hence, the approaches to verify local robustness properties (e.g., Reluplex) can not verify individual fairness [117, 78]. Gopinath et al. proposed a data-driven approach to assess global robustness [78]. The approach requires labeled training data to cluster inputs and then casts the problem as local robustness checking. However, our goal is to verify individual fairness statically, i.e., we do not require training data or an oracle of correct labels.

Definition 1 can be *relaxed* to ensure further fairness of the model. The attributes of x and x' are said to be relaxed when they are not equal. Hence, the protected attribute is always relaxed in the above definition. The constraint ❶ above can also be relaxed on the non-protected attributes so that instead of equality, a small perturbation ϵ is allowed [97]. For example, we can still consider (x, x') as bias instance even if a non-protected attribute (e.g., *age*) differs in a small amount (e.g., 5 years). Note that if a model is fair with respect to Definition 2, it is also fair with respect to Definition 1 but the opposite is not true. Therefore, from the verification perspective, the relaxed query requires stronger certification and it subsumes the basic fairness requirement. In our evaluation, we showed that we can verify custom queries based on the above definitions.

Definition 2 ***(ϵ-Fairness)*** *The model M is individually fair if no two data instances (x, x') in the input domain satisfy:* ❶ $|x_i - x'_i| \leq \epsilon_i$ *,* $\forall i \in A \setminus P$*,* ❷ $x_j \neq x'_j$ *,* $\exists j \in P$*, and* ❸ $M(x) \neq M(x')$*, where ϵ is a small perturbation that limits similarity.*

We also introduce the notion of targeted fairness, which imposes an arbitrary additional constraint on the inputs. For example, the developer might be interested in verifying whether the NN is fair in giving loans to individuals who have at least high school education, works more than 50 hours per week, and occupation is *Sales*.

Definition 3 ***(Targeted Fairness)*** *A target T is a set of arbitrary linear constraints on the inputs of NN such that* $\{(l_i, u_i) \,|\, \forall i \in A,\, l_i \leq x \leq u_i\}$*. Targeted fairness ensures that all valid inputs in the given target satisfy individual fairness.*

The above definition is practical requirement for a specific classification task. Holstein et al. identified that one difficulty in fair software development is to diagnose and audit problems [88]. The authors conducted a survey among 267 ML practitioners, where 62% of the survey respondents indicated that tools to find unfair instances would be very useful. The following response from a data manager conveys the utility of targeted verification and counterexamples [88], which is addressed in this paper.

> *"If an oracle was able to tell me, 'look, this is a severe problem and I can give you a hundred examples [of this problem],' [...] then it's much easier internally to get enough people to accept this and to solve it. So having a process which gives you more data points where you mess up [in this way] would be really helpful."*

6.4 Approach

In this section, we formulate the fairness verification problem and describe our approach to verify the property.

6.4.1 Problem Formulation

The verification problem that we are interested in is, given the precondition on the input x and the NN function M, the output $y = M(\mathrm{x})$ satisfies some postcondition. The preconditions and postconditions are designed to verify the fairness defined in Section 6.3 on a trained model where we have white-box access to the NN. In the definitions, ❶ and ❷ are the fairness preconditions, and ❸ is the fairness postcondition. Suppose, each input x_i is bound by some lower bound lb_i and upper bound ub_i that is obtained from the domain knowledge. Let, M be a classification network with one neuron in the output layer and suppose, Sigmoid function is applied on the output node y to predict the classes. Sigmoid function is given by the equation: $\text{Sig}(y) = 1/(1 + e^{-y})$. Therefore, for any two inputs x, x', the postcondition that needs to be satisfied for fairness is: $M(x) = M(x')$. Here, we encode the negation of the postcondition in the verification query so that the solver provides counterexample with SAT when there is a violation. Next, we compute the weakest precondition (WP) of the postcondition. A WP function gives the minimum requirements needed to be satisfied to assert the postcondition. The WP computation is shown below, which transfers the fairness postcondition to the neurons of the output layer before applying the activation function, Sigmoid.

$$WP(M(x) \neq M(x')) \equiv \text{Sig}(y) \neq \text{Sig}(y')$$

$$WP(\text{Sig}(y) \neq \text{Sig}(y')) \equiv (y < 0 \wedge y' > 0) \vee (y > 0 \wedge y' < 0)$$

Similarly, for other non-linear activation functions, we can compute the WP. Another WP transfer used in our evaluation is for Softmax function: $f(x_i) = e^{x_i}/\Sigma_j e^{x_j}$. The WP of Softmax for binary classification tasks is: $(y_0 > y_1 \Rightarrow y'_0 < y'_1) \wedge (y_0 < y_1 \Rightarrow y'_0 > y'_1)$. Overall, this step makes the verification query simpler as well as the postcondition is reasoned on the network output neurons. Thus, after reducing the fairness postconditions, we have to verify the following constraint to satisfy fairness according to Definition 1.

$$\Big(\overbrace{\bigwedge_{\forall i \in A} lb_i \leq x_i, x'_i \leq ub_i}^{\text{domain constraint, } \phi_x^d}\Big) \wedge \overbrace{\Big(\bigwedge_{\forall j \in A \setminus P} x_j \neq x'_j\Big) \wedge \Big(\bigwedge_{\forall k \in P} x_k = x'_k\Big)}^{\text{fairness precondition, } \phi_x^f} \wedge$$

$$\underbrace{y = M(x), y' = M(x')}_{\text{outputs}} \implies \underbrace{\big(y < 0 \wedge y' > 0\big) \vee \big(y > 0 \wedge y' < 0\big)}_{\text{fairness postcondition, } \phi_y}$$

The above verification formula is defined using two copies of the same NN, its input constraints (x), and output constraints (y). The above constraint also support the Definition 2 and Definition 3. For Definition 2, we update the inequality in ϕ_x^f with $|x_j - x'_j| \leq \epsilon_j$. Additionally, for Definition 2, we use the bounds from target T in ϕ_x^d instead of the original domain constraints.

Unlike local robustness [117, 78], the above fairness constraint requires twice as many input variables (x_i and x'_i) and two copies of networks to obtain outputs (y and y'). Similar to [191], in this chapter, we also considered fairness of NNs trained on structured or tabular data which has been of interest in many recent works [217, 190, 6, 17, 66]. Other than that, the complexity of the constraint depends on the number of neurons in M. In each neuron of the hidden layers, the solver divides the query into two branches: if the weighted sum is non-negative or else, as shown in Equation (6.1). Thus, for n neurons the query divides into 2^n branches, which can not be parallelized [62]. In our approach, we identify the neurons that always remain inactive and remove them to reduce the complexity. Furthermore, we formulate the verification as a SMT based problem so that we can provide counterexample along the certification and verify ϵ-fairness and targeted fairness. Existing abstract interpretation based techniques [191] can not be used towards that goal [80, 77].

6.4.2 Solution Overview

Fairify takes two inputs: trained neural network and fairness verification query. Then it performs three main steps: 1) input partitioning, 2) sound pruning, and 3) heuristic based pruning. An overview of the

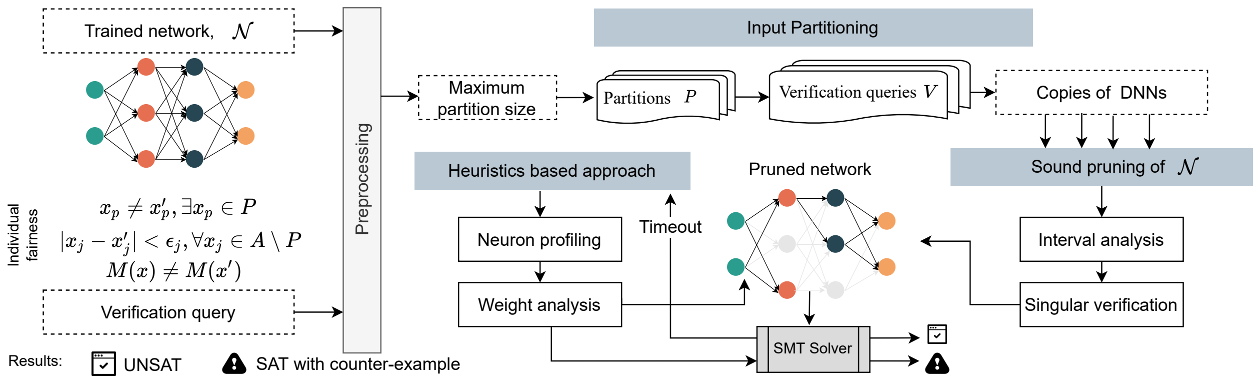

Figure 6.3: The overview of our approach for the fairness verification of neural networks

solution is depicted in Figure 6.3. Before going to step 1, Fairify preprocesses the verification query. It computes the WP of the postcondition to reduce the complexity of the verification formula.

After the preprocessing step, we have precondition defined on the neurons of input layer and postcondition defined on the neurons of the output layer. Then Fairify performs the input partitioning method. The main objective is two-fold: 1) the verification query becomes simpler, i.e., now the solver has to check less input region to prove the constraint; 2) given a smaller partition, the NN exhibits certain activation pattern that we leverage to prune the network. Specifically, one of our key insights is that the NN trained on structured data exhibits sparseness in activation pattern, when we consider a smaller region.

After the input partitioning, Fairify first attempts the sound pruning approach where we use the tightened input bounds for the partitions to compute bound for each neuron. Here, we use the white-box access to the network weights and biases. Then we perform another step of verification on each neuron to prove its activation. This process is applied layer-wise, only one layer at a time. Hence, the checking takes very little time and identifies additional neurons that are inactive. Then inactive neurons are removed from the NN to reduce its size. Removing neurons from the NN largely reduces the complexity of the verification. Finally, Fairify leverages an SMT solver to solve the verification query on the pruned version of the network. Fairify takes a soft-timeout as input parameter. When the soft-timeout is reached without any verification result, Fairify takes the heuristic based pruning approach and attempts to solve the verification query.

In this pruning approach, Fairify uses the network heuristics, e.g., weight magnitudes and their distribution among the neurons. We conduct a simulation on the NN to profile the neurons. If a neuron is never activated, it is a candidate for removal. Then we compare the candidates' distribution of magnitude with that of non-candidates. Thus, we identify additional neurons that are inactive in the given partition. Since this step may prune neurons which are rarely activated, the pruned version can lose small accuracy. Fairify takes a conservative approach to verify the pruned NNs with very little to no loss of accuracy. The approach design also allows to provide **partial verification** result for a subset of the partitions. The goal of verification is to provide SAT/UNSAT for as many partitions as possible within given time budget. Next, we describe three main components of Fairify in detail.

6.4.3 Input Partitioning

Fairify takes the parameter MS, which is the maximum size of an attribute A_i for partitioning. Based on the value of MS, Fairify automatically partitions the input domain into m regions following the Algorithm 1. We first divide each attribute into $\lceil (ub - lb)/MS \rceil$ partitions, and then taking each partition from each attribute, we get the regions, for which the verification query is solved. For each region, we assign a copy of the NN so that different pruning can be applied for each region as well as the query processing can be parallelized. The verification results for the partitions are accumulated as follows.

1. If one partition is SAT, the whole problem is SAT. The counterexample is a violation for the whole verification query. However, it would be desired to check other partitions so that as many counterexamples can be found, which localizes fairness violations and provide insights for repair.

2. If one partition provides UNSAT, the whole problem is not necessarily UNSAT. However, the verification provides guarantee that there are no two inputs possible in the given partition that violates the property. This provides partial certification.

3. To prove that the whole problem is UNSAT, all the partitions have to be UNSAT.

While partitioning, it is important to note that the counterexamples (x and x') cannot lie in the two partitions. In other words, all partitions being UNSAT, if two counterexamples lie in two different partitions, then we might not detect that violation. Therefore, we do not partition the attributes that are relaxed on the inputs. For example, in Definition 1, the protected attribute is always relaxed so that two counterexamples

Algorithm 1 Input partitioning based on domain constraints

```
Input: Attributes $A$, Input domain $\mathbb{I}$, Max-size of attribute $MS$, Query $Q$
Output: Region set $R$
 1: procedure INPUT_PARTITIONER($A$, $\mathbb{I}$, $MS$, $\mathcal{N}$)
 2:     for each attribute $A_i \in A$ do
 3:         $PT_{A_i} = []$
 4:         if $|A_i| > MS$ then
 5:             $low = LB(\mathbb{I}_i)$                          ▷ lower bound of attribute
 6:             $high = UP(\mathbb{I}_i)$                         ▷ upper bound of attribute
 7:             if $Q$ contains $|x - x| \leq \epsilon_i | x, x' \in A_i$ then    ▷ $A_i$ is relaxed
 8:                 $part = [low, high]$                          ▷ no partitioning
 9:             else
10:                 while $low \leq high$ do
11:                     $high = low + MS$
12:                     $part = [low, high]$
13:                     Add $part$ to $PT_{A_i}$
14:                 end while
15:             end if
16:         end if
17:     end for
18:     $R = \{(part) \,|\, part \in PT_{A_i}, \forall A_i \in A\}$
19:     return $R$
20: end procedure
```

can take any two different values. In case of Definition 2, we also ignore partitioning the attribute which is relaxed in addition to the PA. Finally, after the partitioning, we shuffle the partitions to check different parts of the regions in the given timeout. Prioritizing the input partitions to cover more partitions in a quicker time can be a potential future work.

6.4.4 Sound Pruning

This step receives a number of verification problems from input partitioning. Each problem is associated with a copy of NN and the query ϕ. The query is updated from the original query by tightening the bound of each attribute. Now, we attempt to prune the network by removing the neurons that do not impact the prediction of the current copy of the NN.

Static analysis and Pruning of NN. Before deployment, a trained NN can be assessed for certain properties. We obtain the weights and biases of the network to analyze its behavior. Each neuron does not contribute to the decision equally. The value of certain neurons depend on the incoming values, associated weights and bias. Because of the ReLU activation function, many nodes become inactive, i.e., gets a zero valuation whenever the WS is negative. In our approach, we analyze activation pattern to remove those neurons which are always inactive, since they do not affect the NN output. We perform such analysis using specific bounds on the input values. For example, if we can assert $v_3^3 = 0$ for certain bounds on inputs $lb_1 < v_1^1 < ub_1$, $lb_2 < v_1^2 < ub_2$, then removing v_3^3 and the associated edges does not impact the value of v_4^1 and v_4^2. After obtaining the weights and bias, we translate the NN into imperative program representation [80] so that it could be executed symbolically and constraint checker can assert first order formulas. We leverage Numpy arrays and matrix operations to enable tracking the NN structure information (e.g., layer, neuron) and perform NN pruning.

```
def net(x, w, b): # Inputs are Numpy arrays for input data, weight and bias
    # Input layer
    x1 = w[0].T @ x + b[0] # WS calculation using matrix multiplication
    y1 = numpy.maximum(0, x1) # ReLU activation
    # Hidden layer
    x2 = w[1].T @ y1 + b[1]
    y2 = numpy.maximum(0, x2) # ReLU activation
    ... # Output layer
    x_out = w[2].T @ y2 + b[2]
    y = 1 / (1 + math.exp(-x_out)) # Sigmoid activation
    return y # Output of NN
```

We take two steps to find such neurons: 1) interval analysis and 2) individual verification. First, we compute bounds for each neuron in the hidden layers, and then perform layer-wise verification to prove that the given neuron is always inactive. The sound pruning algorithm is presented in Algorithm 2.

Interval analysis We perform bound computation for the neurons in each hidden layer by using the bounds from its preceding layer, as shown in line-18 in Algorithm 2. Here, we used interval arithmetic to compute the bounds. First, we separate the positive and negative incoming weights for a neuron v_i^j. Then we compute its maximum value by multiplying the upper bounds of the incoming neurons with positive weights and lower bounds of the incoming neurons with the negative weights. We do the opposite to get the minimum value of v_i^j. Thus, from the bounds of neurons from its preceding layer, we compute the bounds of the

Algorithm 2 Sound pruning

```
Input: Network $\mathcal{N}$, Weights $W$, Biases $B$, Region $R$
Output: Pruned network $\mathcal{N}'$
 1: procedure SOUND_PRUNING($\mathcal{N}, W, B$)
 2:     candidates = []                              ▷ The candidate neurons for removal
 3:     for hidden neuron $v_i^j$ $in$ Layer $L_i$ do
 4:         lb, ub = NEURON_BOUND($v_i^j, W, B$)
 5:         if ub < 0 then
 6:             candidates.add($v_i^j$)
 7:         else if lb > 0 then
 8:             $\mathcal{N}'$ = Update($\mathcal{N}$)               ▷ Remove ReLU function from $v_i^j$
 9:         else
10:             if INDIVIDUAL_VERIFICATION($v_i^j, L_{i-1}$) then
11:                 candidates.add($v_i^j$)
12:             end if
13:         end if
14:     end for
15:     $\mathcal{N}'$ = Update($\mathcal{N}$)                      ▷ Remove neurons in candidates
16:     return $\mathcal{N}'$
17: end procedure
18: procedure NEURON_BOUND($v_i^j, W, B$)
19:     $p_w$: non-negative incoming weights
20:     $n_w$: negative incoming weights
21:     $p_w = \{w_{i-1,j} \,|\, w_{i-1,j} \geq 0, \forall j \in \{1, ..., |L_{i-1}|\}$
22:     $n_w = \{w_{i-1,j} \,|\, w_{i-1,j} \leq 0, \forall j \in \{1, ..., |L_{i-1}|\}$
23:     $UB(v_i^j) = \Sigma_{w \in p_w} w * UB(v_{i-1}^j) + \Sigma_{w \in n_w} w * LB(v_{i-1}^j) + b_i^j$
24:     $LB(v_i^j) = \Sigma_{w \in p_w} w * LB(v_{i-1}^j) + \Sigma_{w \in n_w} w * UB(v_{i-1}^j) + b_i^j$
25:     return $(UB(v_i^j), LB(v_i^j))$
26: end procedure
27: procedure INDIVIDUAL_VERIFICATION($v_i^j, L_{i-1}$)
28:     precondition $pre = \{LB \leq v_{i-1}^j \leq UB, \forall j \in \{1, ..., |L_{i-1}|\}\}$
29:     postcondition $post = v_i^j < 0$
30:     singular_verification = SMT-Solver($pre, post$)
31:     if singular_verification = UNSAT then
32:         return $true$
33:     end if
34:     return $false$
35: end procedure
```

neurons. The bounds are calculated without considering the activation functions but only the weighted sum. If the weighted sum is always ≤ 0, we can remove the neuron from the NN with the edges connecting to it.

Since the interval arithmetic on multiplication does not lose any accuracy [143], the bounds always hold for the neurons. When we find that the upper bound of v_i^j is ≤ 0 we mark v_i^j for removal. The key intuition for this step is that since we have tightened bounds for the input layer (after partitioning), the bounds of hidden neurons are also tighter. Therefore, we can remove more neurons by applying this step on the partitioned regions. However, the bounds of many neurons, especially in deep layers, may not be tight enough to prove it is inactive. So, we apply the next step of individual verification on each neuron.

Individual verification We formulate verification query for each hidden neuron to further prove whether it is inactive. The precondition for the verification query for neuron v_i^j consists of the bounds each v_{i-1}^j, and the postcondition is $v_i^j < 0$. Then we leverage the SMT solver to prove that the neuron is inactive. We run these individual verification queries on neurons in the increasing order of the layers. So, all the neurons of one layer is verified before going to the next layer. Similar to the bound analysis, the verification query does not include any non-linear activation functions i.e., ReLU. In addition, the query is designed with the precondition that contains values of only the immediate preceding layer. Furthermore, we apply individual verification query only on the neurons that are not already pruned by interval analysis. Therefore, the individual verification queries are faster and can find more inactive neurons that were not found in the previous step.

6.4.5 Heuristic Based Pruning

We investigated the real-world models used in the fairness problems on structured data, and we found that the datasets are far sparse from problem that involve high dimensional data such as image classification or natural language processing. For example, each instance in the MNIST dataset is (28, 28) Numpy array, whereas a popular Adult Census dataset contains data instances each with the shape (1, 13). It causes the trained NN to have many neurons that are never activated or only activated for a specific region. Applying the methods in the previous stage, we could detect some provably guaranteed inactive neurons in the NN. But there can be further such neurons which are not activated for the given region in practice. How can we detect those neurons?

A variety of network heuristics has been studied for different purposes such as quantization, efficiency, accuracy, etc. [55, 82, 83], which can be performed in different stages of the development e.g., before or during training. Inspired by that, we proposed a conservative approach of NN pruning for fairness verification based on heuristics. Our goal is to reduce the network size without losing accuracy as much as possible.

Algorithm 3 Heuristics based pruning

```
Input: Network $\mathcal{N}$, Domain $\mathbb{I}$, Simulation size $S$, Tolerance $T$
Output: Pruned network $\mathcal{N}'$
 1: procedure HEURISTIC-PRUNE($\mathcal{N}$, $\mathbb{I}$, $S$)
 2:     $candidates, Mag^+, Mag^-$ = NEURON-PROFILER($\mathcal{N}, S$)
 3:     $Dist_{candidate}$                              ▷ Distribution of the candidates
 4:     $Dist_{noncandidate}$                           ▷ Distribution of the non-candidates
 5:     for $v_i^j$ in hidden layer $L_i$ do
 6:         check candidates and noncandidates distribution differs
 7:         if $Mag^+(v_i^j) < T$-level of $Dist_{noncandidate}$ then
 8:             Remove $v_i^j$ from $M$
 9:         end if
10:     end for
11: end procedure
12: procedure NEURON-PROFILER($\mathbb{I}$, $S$)
13:     $\mathcal{D} = \{X | X_i \in \mathbb{I}$ and $|X| = S\}$
14:     $candidates = \{v_i^j \mid v_i^j \in L\}$               ▷ L is a hiden layer in $M$
15:     for $X_i \in \mathcal{D}$ do
16:         Run $\mathcal{N}(X_I)$ to record heuristic
17:         for hiddent neurons $v_i^j$ in $\mathcal{N}$ do
18:             if $v_i^j$ is active then
19:                 $candidates.remove(v_i^j)$
20:             end if
21:             if $v_i^j \geq 0$ then                          ▷ Positive magnitude
22:                 $Mag^+ = |v_i^j|$
23:             else                                        ▷ Negative magnitude
24:                 $Mag^- = |v_i^j|$
25:             end if
26:         end for
27:     end for
28:     return $candidates, Mag^+, Mag^-$
29: end procedure
```

The algorithm for heuristic based pruning is shown in Algorithm 3. For learning the heuristics of the network, we run the neuron profiler. The objective is to get the distribution of magnitudes of each neuron.

First, we create a simulated dataset $\mathcal{D}$ of size S by randomly choosing valid inputs from attribute domain. Then the network M is run S times to record the values for each neuron. In practice, we used $S = 1000$, which allows to separate most of the candidates and non-candidates. Initially, all the neurons are selected as candidates for removal. If a neuron gets non-zero value for at least one execution of the simulation, we remove that from the candidate list. Thus, after recording the values of both the candidates and non-candidates, we confirm that whether the distribution of positive magnitude differs significantly. The intuition is that even if the neuron in the candidate is inactive for all the executions, it might get activated for some valid input data. However, the magnitude distribution of the neurons suggests that whether it is active or not. In practice, we observe a significant difference in their distribution. For example, we observed that in the first hidden layer of the NNs, the mean and median of the positive magnitude of non-candidates is at least 10 times larger.

The above heuristic allows to compare the magnitude of each candidate neuron and conservatively select it for removal. Fairify takes a tolerance level as input for this comparison. In our evaluation, we used 5% as the tolerance level. Fairify selects the candidates which has a magnitude less than the 5-percentile of the non-candidate neurons. The weight magnitude difference is more prominent in deep layers than the shallow ones. So, we do layer-wise comparison and candidate selection based on the heuristics.

One might argue that since this pruning does not provably guarantee the same outcome as the original NN, the verification result can be inaccurate. However, the idea here is to use the pruned version of the NN in production as opposed to the original version. If we certify the pruned NN and observe little or no accuracy decrease, then the pruned and verified model itself can be used in the production. Thus, we can preserve the sanity of the verification results. Our evaluation shows that it is possible to carefully select the heuristic so that NN is pruned conservatively, which affects accuracy negligibly but enables faster verification.

6.5 Evaluation

In this section, first, we discuss the experimental details and then answer the research questions regarding the utility, scalability, and performance of our approach.

6.5.1 Experiment

Benchmarks The verification benchmark is a crucial part of the evaluation. The performance of the verifier depends on the benchmark models. Shriver et al. demonstrated the need for consistent benchmarks for verifying different properties of NN [176]. As an example, NNs with the same structure (e.g., number of layers and neurons) would be very different if trained for *image classification* or *loan approval*. For the latter, the NN is trained on structured data and exhibits unique neural characteristics for which Fairify is designed. We created a comprehensive benchmark for fairness verification of NN models collected from prior works and practice.

We evaluated Fairify on three popular fairness datasets i.e., Bank Marketing (BM), Adult Census (AC), and German Credit (GC) [217, 66, 5, 29]. The benchmark models (Table 6.1) are collected from four different sources. First, we followed the methodology of Biswas and Rajan to collect real-world NNs from Kaggle [27]. We searched all the notebooks under the three datasets in Kaggle and found 16 different NNs. Second, Zhang et al. used 3 NNs trained on the aforementioned datasets [217], which we included in our benchmark. Third, Udeshi et al. [190] evaluated fairness testing on one NN architecture, which is further used by Aggarwal et al. [5] on the three datasets. We found that two of these three models are also implemented in Kaggle notebooks. Finally, NN models AC8-12 were created by [191, 131] for dependency fairness certification. Thus, we created a fairness benchmark of 25 NNs. The networks used for these fairness problems are fully connected with ReLU activation functions, which is also observed by many prior works [217, 6, 191, 131]. In addition, the accuracies of the models are very high compared to the other ML classifiers trained on these datasets. The models and datasets are placed into our replication package to make the tool self-contained. The details of the datasets are as follows:

Bank Marketing. The dataset contains marketing data of a Portuguese bank which is used to classify whether a client will subscribe to the term deposit [133]. It has 45,000 data instances with 16 attributes and *age* is considered as the protected attribute.

German Credit. This dataset contains 1000 data instances of individuals with 20 attributes who take credit from a bank [87]. The task is to classify the credit risk of a person, whereas *sex* and *age* are considered as the protected attributes.

Table 6.1: The neural network benchmark for fairness verification

Dataset	Model	Source	# Layers	# Neurons	Accuracy %
Bank Marketing	BM1	Kaggle	4	97	89.20
	BM2	Kaggle	4	65	88.76
	BM3	Kaggle, [5, 190]	3	117	88.22
	BM4	Kaggle	5	318	89.55
	BM5	Kaggle	4	49	88.90
	BM6	Kaggle	4	35	88.94
	BM7	Kaggle	4	145	88.70
	BM8	[217]	7	141	89.20
German Credit	GC1	Kaggle	3	64	72.67
	GC2	[5, 190]	3	114	74.67
	GC3	Kaggle	3	23	75.33
	GC4	Kaggle	4	24	70.67
	GC5	[217]	7	138	69.33
Adult Census	AC1	Kaggle	4	45	85.24
	AC2	Kaggle, [5, 190]	3	121	84.70
	AC3	Kaggle	3	71	84.52
	AC4	Kaggle	4	221	84.86
	AC5	Kaggle	4	149	85.19
	AC6	Kaggle	4	45	84.77
	AC7	[217]	7	145	84.85
	AC8	[191, 131]	4	10	82.15
	AC9	[191, 131]	6	12	81.22
	AC10	[191, 131]	6	20	78.56
	AC11	[191, 131]	6	40	79.25
	AC12	[191, 131]	11	45	81.46

Adult Census. This dataset contains United States census data of 32,561 individuals with 13 attributes [125]. The task is to predict whether the person earns over $50,000. Here, *race* and *sex* are considered as the protected attributes.

Experiment setup Fairify is implemented in Python and the models are trained using Keras APIs [191]. Following the prior works in the area [6], we used Z3 [53] as the off-the-shelf SMT solver for manipulating the first-order formulas. However, other SMT solvers can also be used, since our technique of input partitioning and network pruning can work independent of any SMT solver. The experiments are executed on a 4.2 GHz Quad-Core Intel Core i7 processor with 32 GB memory.

Input. Fairify takes the trained NN model, input domain, the verification query, maximum partition size, and timeout as inputs. The trained models are saved as `h5` files, from which Fairify extracts the necessary information e.g., weight, bias, structure. The verification query includes the name of the protected attributes and relaxation information of the other attributes (if any).

Output. Fairify provides verification result for each partition separately. The results include verification (SAT/UNSAT/UNKNOWN), counterexample (if SAT), and pruned NN for the given partition.

6.5.2 Results

We answered three broad research questions to evaluate Fairify:

- RQ1: What is the **utility** of our approach?
- RQ2: Is Fairify **scalable** to relaxed fairness queries?
- RQ3: What is the **performance** with respect to time and accuracy of the approach?

6.5.2.1 Utility

First, we evaluated the baseline fairness verification results and then compared with our approach, which is shown in Table 6.2. Since GC and AC contained multiple protected attributes (PA), we setup multiple verification for each of those NNs. However, the verification results are consistent over different PAs. Hence, we showed result for one PA for each dataset in Table 6.2, and the whole result is also in our supplementary material[2]. In this RQ, we evaluated the individual fairness defined in Equation (1), which requires two individuals with same attributes (except the protected attribute) are classified to the same class. We will discuss the verification of the relaxed fairness constraints Equation (2) and Equation (3) in RQ2.

For the baseline verification, we encoded fairness property into satisfiability constraints, and then passed the *original NN* and constraints to the SMT solver. In addition, we attempted to verify the original NN with input partitioning. On the other hand, Fairify used the method of input partitioning and NN pruning to demonstrate the improvement over the baseline. For the baseline, we set a timeout of 30 hours for the solver

[2]https://github.com/anonymous-authorss/Fairify/blob/main/Appendix-Result.pdf

Table 6.2: Fairness verification results for the neural networks. Experiment setup: soft-timeout 100s, hard-timeout 30m, max-partition size: 100 (BM, GC), 10 (AC). BL: Baseline (w/ P - with Partitioning), Ver: Verification, #P: number of partitions, Cov: Coverage, US: UNSAT, UNK: UNKNOWN #H-At: number of times heuristic pruning attempted, C: average **c**ompression ratio – (S): sound pruning, (H): heuristic pruning, Average time (second) ⏱ – SV: sound verification, HV: heuristic verification.

PA	Model	BL	B w/ P	Ver	#P	Cov %	SAT	US	UNK	#H	#HS	C(S)	C(H)	SV ⏱	HV ⏱	Total ⏱
Age	BM1	UNK	UNK	SAT	75	13.14	9	58	8	9	1	0.90	0.01	14.04	11.00	25.40
	BM2	UNK	UNK	SAT	141	26.08	30	103	8	9	1	0.85	0.01	7.76	5.83	13.84
	BM3	UNK	UNK	SAT	139	26.47	27	108	4	6	2	0.96	0.00	9.78	3.42	13.49
	BM4	UNK	UNK	SAT	37	6.08	1	30	6	6	0	0.91	0.03	17.10	16.29	49.98
	BM5	UNK	UNK	SAT	510	99.61	114	394	2	7	5	0.83	0.00	2.29	0.54	2.96
	BM6	UNK	UNK	SAT	510	100.00	156	354	0	3	3	0.76	0.00	1.17	0.08	1.36
	BM7	UNK	UNK	SAT	124	23.33	62	57	5	9	4	0.92	0.01	8.31	5.10	14.89
	BM8	UNK	UNK	SAT	23	3.14	1	15	7	10	3	0.79	0.04	46.23	31.64	79.08
Sex	GC1	UNK	UNK	SAT	31	13.43	27	0	4	6	2	0.76	0.01	41.78	15.98	58.18
	GC2	UNK	UNK	SAT	11	1.99	4	0	7	9	2	0.78	0.04	97.18	75.67	174.18
	GC3	UNK	UNK	SAT	201	100.00	195	6	0	1	1	0.69	0.00	1.29	0.24	1.63
	GC4	UNK	UNK	SAT	201	100.00	2	199	0	1	1	0.63	0.00	0.73	0.14	0.97
	GC5	UNK	UNK	UNK	12	1.49	0	3	9	9	0	0.59	0.03	75.27	75.43	151.53
Age	GC1	UNK	UNK	SAT	46	20.40	41	0	5	8	3	0.75	0.01	29.72	12.96	43.19
	GC2	UNK	UNK	SAT	13	3.98	8	0	5	10	5	0.82	0.04	88.11	52.42	141.86
	GC3	UNK	UNK	SAT	201	100.00	195	6	0	1	1	0.69	0.00	1.41	0.22	1.74
	GC4	UNK	UNK	SAT	201	100.00	2	199	0	1	1	0.63	0.00	0.73	0.16	1.00
	GC5	UNK	UNK	UNK	13	1.99	0	4	9	9	0	0.62	0.03	69.49	69.73	139.73
Race	AC1	UNK	UNK	SAT	29	0.14	8	15	6	8	2	0.64	0.04	40.73	23.50	64.43
	AC2	UNK	UNK	SAT	15	0.04	4	3	8	10	2	0.82	0.04	72.62	54.86	128.45
	AC3	UNK	UNK	SAT	23	0.11	16	1	6	10	4	0.75	0.02	52.75	32.33	85.40
	AC4	UNK	UNK	UNK	8	0.00	0	0	8	8	0	0.67	0.20	100.57	100.08	241.70
	AC5	UNK	UNK	SAT	10	0.02	3	0	7	9	2	0.71	0.13	98.14	74.06	180.41
	AC6	SAT	SAT	SAT	19	0.07	6	5	8	9	1	0.49	0.05	55.70	44.43	100.37
	AC7	UNK	UNK	UNK	14	0.04	0	7	7	10	3	0.59	0.11	78.22	51.09	132.28
	AC8	UNK	UNK	SAT	101	0.63	82	19	0	1	1	0.22	0.00	17.59	0.13	17.84
	AC9	UNK	UNK	SAT	741	4.63	399	342	0	4	4	0.19	0.00	2.29	0.02	2.44
	AC10	UNK	UNK	SAT	20	0.09	6	8	6	10	4	0.27	0.05	62.07	32.82	95.07
	AC11	UNK	UNK	UNK	9	0.00	0	0	9	9	0	0.11	0.02	100.13	100.12	200.66
	AC12	UNK	UNK	UNK	16	0.04	0	7	9	9	0	0.29	0.01	57.56	56.35	114.19
Sex	AC1	UNK	UNK	SAT	24	0.11	4	13	7	10	3	0.65	0.06	43.46	32.52	76.18
	AC2	UNK	UNK	SAT	17	0.07	3	8	6	8	2	0.83	0.02	63.50	41.78	106.05
	AC3	UNK	UNK	SAT	31	0.17	27	0	4	8	4	0.75	0.02	44.63	16.76	61.72
	AC4	UNK	UNK	SAT	9	0.01	2	0	7	8	1	0.78	0.10	98.98	82.21	205.17
	AC5	UNK	UNK	SAT	9	0.00	0	0	9	9	0	0.77	0.09	100.20	100.11	205.50
	AC6	UNK	UNK	SAT	22	0.09	8	6	8	10	2	0.57	0.05	46.65	39.40	86.26
	AC7	UNK	UNK	UNK	13	0.04	0	7	6	11	5	0.63	0.08	86.51	52.15	141.19
	AC8	UNK	UNK	SAT	103	0.64	80	23	0	2	2	0.24	0.00	16.99	0.42	17.53
	AC9	UNK	UNK	SAT	332	2.08	161	171	0	1	1	0.20	0.00	5.28	0.17	5.56
	AC10	UNK	UNK	SAT	31	0.18	10	18	3	10	7	0.32	0.03	44.93	14.98	60.07
	AC11	UNK	UNK	UNK	9	0.00	0	0	9	9	0	0.12	0.04	100.15	100.13	200.73
	AC12	UNK	UNK	UNK	21	0.08	0	12	9	9	0	0.34	0.01	44.90	42.91	88.07

and run the verification for the models in our benchmark. Only one model (AC6) could be verified within the timeout. For the other models, the solver reported UNKNOWN i.e., the model could not be verified within the time limit. Furthermore, we try the baseline verification with input partitioning and run the queries on the NNs but we get the same verification result as the baseline. The main takeaway is that the difficulty in verifying fairness lies in the complex structure of the NNs. Only input partitioning reduces the input space to be verified, but that does not reduce the complexity of the NN under verification. Fairify combines partitioning and pruning to enable network complexity reduction, which in turn made the verification feasible.

How effective is our approach to verify fairness of NN? We presented the verification results of 25 NNs in Table 6.2. We found that Fairify produces verification results very quickly compared to the baseline. For each model, we run the verification task for 30 minutes (hard-timeout). Since we have divided the single verification into multiple partitions, we set 100 seconds as the soft-timeout for the SMT solver, which means that the solver gets at most 100 seconds to verify. When the result of the sound verification is UNKNOWN, then Fairify attempts the heuristic based prunning and runs the SMT solver for another 100 seconds. The goal of using a short soft-timeout is to show the effectiveness of our approach over baseline.

The results show that 19 out of 25 models were verified within the 30 minutes timeout. The models that could not be verified in that time period were considered again with scaled experiment setup in RQ2. Fairify takes the maximum size of an attribute (MS) as an input to automatically partition the input region. The user can select MS based on the range of the attributes in the dataset. The timeout and MS can be configured based on the budget of the user. For BM and GC models, we used 100 as MS, and for AC models we used 10 as MS. To that end, Fairify divided input region into 510, 201, and 16000 partitions using Algorithm 1. Here is an example verification task from our evaluation:

Example: While verifying AC3 (*race* as PA), Fairify takes the following partition as a sub-problem. First, it attempts sound pruning and achieves 86.27% compression. Then it runs verification query for 100 seconds and reports SAT with the counterexamples **C1** and **C2** in 21.47 seconds. Note that, these are two input examples for which the NN is not fair with respect to *race*. Here, the two individuals had the same attributes except for *race* but were classified as *bad* and *good* credit-class, respectively.

workclass: [0, 6], marital-status: [0, 6], relationship: [0, 5], race: [0, 4], sex: [0, 1], age: [80, 89], education: [0, 9], education-num: [11, 16], occupation: [0, 9], capital-gain: [10, 19], capital-loss: [0, 9], hours-per-week: [51, 60], native-country: [30, 39]

C1: $[89, 6, 9, 14, 1, 0, 0, 0, 1, 14, 8, 59, 39]$

C2: $[89, 6, 9, 14, 1, 0, 0, 0, 0, 14, 8, 59, 39]$

Similar to the above example, Fairify checks a number of partitions for each model and reports the results. Depending on the complexity of each NN, Fairify could complete verification for a certain number of partitions in the given timeout period. Whenever we get SAT for at least one partition, the whole verification is SAT. Fairify also reports the counterexample when it reports SAT. We included the detailed results for each partition including all generated counterexamples in our replication package.

Is the NN pruning effective for verifying fairness? The baseline models cannot be verified in a tractable amount of time. After being able to prune the models significantly, we could verify them in a short time period. We computed the amount of pruning applied to the models. For the original NN M and pruned version M', compression ratio is calculated using the following formula: $1 - |M|/|M'|$, where $|M|$ is the number of neurons in M. The average compression percentage in Table 6.2 shows that Fairify could reduce the size of NN highly in all the models. Heuristic based pruning is only applied when Fairify cannot get verification result within the soft-timeout. Furthermore, heuristic based pruning is applied on the already pruned version of the NN. Therefore, compression percentage for heuristics based pruning is less. We found that 13.98% times Fairify attempted heuristic pruning, 19.38% of those times Fairify provided SAT or UNSAT result, meaning that the additional pruning done by the method helped to complete the verification. In the example showed above, Fairify did not attempt a heuristic based pruning, since it got the result in sound pruning step.

Can Fairify be used to localize fairness defects? For ML models, it is difficult to reason or find defects since the model learns from data. In fairness problems, oftentimes the model learns from biased data or augments the bias during training. If we can filter the input domain where the model is unfair, then it would guide fairness repair. For example, in model BM3, Fairify provides SAT for 27 partitions and UNSAT for 108 partitions. The developer can leverage the verification result to further improve the training data and retrain the network. Another novelty is that since Fairify verifies multiple copies of pruned NN, the developer may choose to deploy those pruned versions into production. When an input comes to the system, the software can choose to use the verified copies of NN. Qualitatively, Fairify provides the following two main utilities for fairness verification.

1) Tractability and speedup: The results showed that the verification for the NN becomes tractable when our approach is applied. Furthermore, the partition size and timeout can be tuned based on the complexity of the NN. In our evaluation, we used a short timeout of 30 minutes [143]. The verification has been possible for most of the models in this time because of successful NN pruning. In addition, as soon as the first SAT is found, the developer may choose to stop running verification for remaining partitions.

2) Partial verification: Even after getting SAT for one partition, our evaluation continued verification for other partitions, which essentially provides partial verification. For example, partitions with UNSAT imply that the NN is fair for that input region. Similarly, SAT with the counterexamples for a partition can be used towards repairing the NN, which is a potential future work.

6.5.3 Scalability

In RQ1, we have set a small timeout and could verify 19 out of 25 models. In this RQ, we set a scaled experiment setup to further verify the remaining 6 models. We noticed that these 6 NNs are complex because of more number of layers and neurons. This time we used a hard-timeout of 1 hour and soft-timeout of 200 seconds, essentially doubling the timeouts. We also reduced the maximum size of the partition (MS). The results are shown in Table 6.3. We have verified the 3 out of the 6 models within that 1 hour. The results demonstrates the scalability of our approach for more complex NNs. We found that for some partitions, the NN compression ratio of Fairify is more, and hence the SMT solver could verify quickly. So, it would be an interesting future work to prioritize the partitions to verify for efficiency. Next, we showed whether our approach can verify complex verification queries for all the models.

Can Fairify verify relaxed and targeted verification queries? To answer this question, we created relaxed and targeted fairness queries according to Definition 2 and Definition 3 respectively. The verification results for the relaxed queries are presented in Table 6.4. First, for the **relaxation** of individual fairness, we

Table 6.3: Verification of NN with scaled experiment setup. Soft-timeout 200s, hard-timeout 60m, max-partition: 10 (for BM, GC), and 6 (for AC). Ver: Verification, #P: number of partitions, H: number of times heuristic pruning attempted, C: average **c**ompression ratio – (S): sound pruning, (H): heuristic pruning, Average time (second) ⏱ – SV: sound verification, HV: heuristic verification.

Model	**Ver**	**#P**	**#sat**	**#unsat**	**#unk**	**H**	**HS**	**C(S)**	**C(H)**	**SV ⏱**	**HV ⏱**	**Total ⏱**
BM8	SAT	48	4	37	7	10	3	.85	.02	47.64	33.46	81.68
GC5	UNK	12	0	3	9	9	0	.60	.03	150.25	150.55	301.83
AC4	SAT	24	15	4	5	9	4	.92	.02	96.65	56.17	160.26
AC7	UNK	21	0	15	6	9	3	.77	.02	111.11	65.64	178.12
AC11	UNK	9	0	0	9	9	0	.21	.02	200.23	200.37	401.18
AC12	SAT	35	3	26	6	9	3	.41	.01	63.47	43.93	107.71

define small perturbation (ϵ) on the non-protected attributes so that two individuals are considered similar even if they are not equal in any non-protected attributes. Those two individuals still have to be classified to the same class to ensure fairness. Therefore, the relaxation on the queries impose stricter fairness requirement. We created six different such queries run Fairify on all the models. In each query, we selected an additional non-protected attribute, which was relaxed. For example, a verification query for AC is (ϕ_{r31}): *Is the NN fair with respect to race, where any two people are similar irrespective of their marital status?* We found that 21/25 for ϕ_{r1} and 22/25 for ϕ_{r2} were verified within one hour. Compared to Table 6.2, the number of SAT found is more since ϵ-fairness is a stricter requirement and creates possibility of more counterexamples. Second, for the **targeted** verification, we created six other fairness queries that targets a specific population. For example, ϕ_{t32} verifies that *whether the NNs in AC are fair for people who has bachelor or doctorate education*. We also find that Fairify can verify most of the models in a quick time. Similar to the scaled experiment setup in Table 6.3, in this experiment, we used a smaller partition size. Although the queries are more complex, Fairify could provide counterexample or certification for many partitions. The number of partitions that are verified depends on the relaxed attribute and the size of the NN.

6.5.4 Performance

How quickly the verification is done and what is the overhead? Depending on the verification query and the model, the verification time varies. Table 6.2 presents the average time taken for the partitions of each model. We showed the time taken by the SMT solver to output a result in the sound verification phase, and in

Table 6.4: Verification of neural networks for relaxed and targeted fairness queries. **Relaxed fairness queries:** ϕ_{r11}: duration < 5, ϕ_{r12}: job < ∞, ϕ_{r21}: credit-amount < 100, ϕ_{r22}: foreign-worker < ∞, ϕ_{r31}: marital-status < ∞, ϕ_{r32}: age<5, **Targeted fairness queries:** ϕ_{t11}: personal-loan & profession:entrepreneur, ϕ_{t12}: previous-marketing:yes, ϕ_{t21}: #credits=2, ϕ_{t22}: foreign-worker & credit-purpose:education, ϕ_{t31}: 30≤age≤35, ϕ_{t32}: education:bachelor or doctorate

	Res	ϕ	**BM1**	**BM2**	**BM3**	**BM4**	**BM5**	**BM6**	**BM7**	**BM8**	ϕ	**GC1**	**GC2**	**GC3**	**GC4**	**GC5**	ϕ	**AC1**	**AC2**	**AC3**	**AC4**	**AC5**	**AC6**	**AC7**	**AC8**	**AC9**	**AC10**	**AC11**	**AC12**
Relaxed	Ver		SAT	SAT	SAT	SAT	SAT	SAT	SAT	SAT		SAT	SAT	SAT	SAT	UNK		SAT	SAT	SAT	SAT	SAT	SAT	UNK	SAT	SAT	SAT	UNK	UNK
	#P		333	566	1600	132	4915	8671	590	112		111	32	2697	9695	9		216	75	129	41	37	177	66	639	3991	216	31	123
	#sat	ϕ_{r11}	40	65	233	10	731	1484	168	11	ϕ_{r12}	107	24	2026	185	0	ϕ_{r13}	9	8	34	5	3	28	0	218	483	19	0	0
	#unsat		279	493	1358	111	4180	7187	411	87		0	2	671	9510	0		193	55	86	22	18	133	53	418	3508	192	13	107
	#unk		14	8	9	11	4	0	11	14		4	6	0	0	9		14	12	9	14	16	16	13	3	0	5	18	16
	Ver		SAT	SAT	SAT	SAT	SAT	SAT	SAT	SAT		SAT	SAT	SAT	SAT	UNK		SAT	SAT	SAT	SAT	SAT	SAT	UNK	SAT	SAT	SAT	UNK	SAT
	#P		968	1514	2768	157	7601	13779	1340	188		116	30	3175	8361	20		210	74	124	30	27	117	44	402	4811	164	18	53
	#sat	ϕ_{r21}	65	110	208	10	583	1200	187	6	ϕ_{r22}	106	16	2343	79	0	ϕ_{r23}	47	20	84	13	11	47	0	182	1294	34	0	1
	#unsat		890	1396	2550	143	7014	12579	1151	167		1	2	832	8282	2		157	45	35	5	3	56	31	220	3517	128	0	36
	#unk		13	8	10	4	4	0	2	15		9	12	0	0	18		6	9	5	12	13	14	13	0	0	2	18	16
Targeted	Ver		SAT	SAT	SAT	SAT	SAT	SAT	SAT	SAT		SAT	SAT	SAT	SAT	UNK		SAT	SAT	SAT	SAT	SAT	SAT	UNK	SAT	SAT	SAT	UNK	UNK
	#P		261	546	1541	92	5937	12233	544	90		162	32	3810	11067	18		223	67	115	25	23	111	38	448	6800	108	18	68
	#sat	ϕ_{t11}	34	61	208	9	881	1939	157	8	ϕ_{t12}	153	17	2786	96	0	ϕ_{t13}	55	16	75	7	8	52	0	179	1624	23	0	0
	#unsat		216	479	1322	72	5050	10294	374	67		0	2	1023	10971	0		160	37	32	3	1	47	26	266	5176	81	0	50
	#unk		11	6	11	11	6	0	13	15		9	13	1	0	18		8	14	8	15	14	12	12	3	0	4	18	18
	Ver		SAT	SAT	SAT	SAT	SAT	SAT	SAT	SAT		SAT	SAT	SAT	SAT	UNK		SAT	SAT	SAT	SAT	SAT	SAT	UNK	SAT	SAT	SAT	UNK	SAT
	#P		432	683	1987	108	6474	13855	986	89		187	46	5957	13426	18		281	85	119	35	33	119	43	677	6781	164	18	59
	#sat	ϕ_{t21}	57	76	270	10	904	2224	285	11	ϕ_{t22}	175	28	3534	63	0	ϕ_{t23}	61	18	79	14	16	51	0	309	1765	36	0	1
	#unsat		361	602	1705	87	5569	11631	693	64		4	7	2422	13363	0		214	58	33	8	4	57	26	368	5016	125	0	41
	#unk		14	5	12	11	1	0	8	14		8	11	1	0	18		6	9	7	13	13	11	17	0	0	3	18	17

heuristic verification phase. We also calculated the total time taken to complete verification, which includes the partitioning and pruning discussed in Section 6.4. The results show that the additional time taken for the three steps in our approach is negligible with respect to the time taken by the SMT solver. The partitioning of inputs is a one time step. For each partition, we apply pruning once or twice. However, pruning is static operation without any complex constraint solving. The only part of our pruning steps that take more time compared to other steps is the individual verification of each neuron which uses the SMT solver. But in that verification, only one layer is considered at a time, activation functions are excluded, and the number of constraints is at most the number of hidden neurons in a layer. Therefore, excluding the time taken by the SMT solvers to solve the final constraints, our approach did not take more than 10 seconds for any model.

What is the performance of Fairify compared to the related work? As described in Section 6.3, individual fairness verification of NN can not be done with existing robustness checkers [78, 117]. Shriver et al. proposed a framework called DNNV [176] that incorporates the state-of-the-art NN verifiers, e.g., ReluPlex, Marabou, etc. We tried verifying the fairness queries using DNNV verifiers, however, those can verify queries with a single network input variable [176, 177]. We verified the models (AC8-12) from Libra [191, 131] shown in Table 6.2 and Table 6.4. Fairify could verify all the models except AC11-12 for two out of four queries. The overall coverage is less than that reported by [131] because Fairify verifies a different property, the configuration is lower, and experiment setup is different. Fairify divides input domain based on the configuration and verifies each partition but Libra computes abstract domain and projects into the input space to find biased region. So, the precision of Libra depends on the chosen abstract domain. On the other hand, Fairify can be configured for arbitrary queries, partial verification, and additionally we provide counterexample and pruned NN as output. Therefore, Fairify can be more appropriate for defect localization or repair. One limitation of Fairify is that when the NN is deep and wide at the same time, the pruning ratio is less, and the SMT solver may return UNK in the given timeout. However, we can use less conservative approach in heuristic pruning to circumvent the problem, dynamically tuning the configuration of Fairify, which is a potential future work.

What is the accuracy loss of heuristic-based pruning? We calculated the accuracy of the pruned NNs for each partition. When no heuristic based pruning is done, there is no accuracy loss. However, when Fairify applies heuristic based pruning there might be accuracy loss compared to the original NN. A pruned version of NN is assigned for a given input partition. The training or test data may not have instances in the given partition. So, we generated synthetic data to evaluate accuracy of each pruned NN. We took a conservative approach for the heuristic based pruning. Therefore, there was no accuracy loss for the pruned NN. Note that even if there is a small accuracy reduction for a heuristic based approach, the developer may choose to deploy the pruned version of the NN as opposed to the original one.

6.6 Related Works

Verification of neural network has been studied in many application domains and property of interest. The robustness of NN has gained a lot of attention for safety-critical applications e.g., autonomous vehicles [134, 44, 75]. Research showed that NN can be fooled by small perturbations to data instances i.e., adversarial input [184, 134, 37, 75, 214]. Section 6.3 further describes NN verification and how it compares for the fairness property. Katz et al. proposed efficient SMT solving algorithms to provide formal guarantee in NN [116, 118]. Also, verification algorithms have been proposed that uses off-the-shelve SMT solver [61, 92]. Research has also been conducted to compute tight bounds of the neurons and provide probabilistic guarantees for some properties [183, 199, 178]. With the extensive use of NN, many recent work focused on new types of properties and both static and dynamic analysis [143, 37, 130, 186, 93].

With the increasing need to ensure fairness of AI based systems [27, 29, 216, 90, 40], many recent works focused on the fairness testing and verification of ML models [66, 190, 5, 217, 6, 17, 97, 218]. Especially, some recent works proposed individual fairness testing on NN [218, 217]. While input test generation has been helpful to find fairness violations, verification is more difficult since it proves certain property for all possible inputs. Probabilistic verification techniques have been proposed to verify group fairness property [6, 17]. Recently, John et al. proposed individual fairness verification approach for three different kinds of ML models [97]. Urban et al. proposed Libra to provide dependency fairness certification for NN using abstract interpretation [191]. They used forward and backward analysis to compute input region that is fair or unfair using different abstract domains e.g., Boxes, Symbolic, Deeppoly. Mazzucato and Urban extended Libra with another abstract domain (Neurify) and implemented automated configuration for scalability and parallelization [131]. Section 5.1 also describes the existing verification techniques with comparison to our work.

6.7 Summary

In this work, we addressed the fairness verification problem of NN. Given the complex structure, providing formal guarantee to the NN is difficult. Previous works considered NN certification for different properties. We are the first to verify individual fairness property for NN that has been shown to be useful in fairness testing in prior work. Our technique, Fairify, can verify various relaxations of individual fairness and provide counterexamples, which has not been tackled before. Fairify uses lightweight techniques to reduce the problem into multiple sub-problems and prune the NN to reduce the complexity of the verification task. Then Fairify applies bound analysis and individual verification to provably prune the neurons that always remain inactive for a given partition. In addition, we conducted neuron profiling to observe their heuristic and prune further. Fairify showed much improvement over the baseline in verifying many real-world NN. The result of Fairify can be leveraged in fairness testing for guided test case generation. Also, the counterexamples can be used to repair the NN in interactive verification setting. Finally, novel static analysis can be proposed in the future to prune the network further and make verification more efficient.

CHAPTER 7. CONCLUSION AND FUTURE WORK

Fairness in machine learning has received much attention recently. This dissertation shows many possibilities to apply software engineering and programming language techniques in understanding and reasoning fairness properties of ML models. We have found what kind of bias is more common and how they could be addressed. Our study also suggests that further SE research and library enhancements are needed to make fairness concerns more accessible to developers. Then we investigated how the data preprocessing stages affect fairness in a pipeline. We have proposed method to measure fairness of the data transformers. We have also showed that existing bias can be mitigated by instrumenting the stages. Finally, we have proposed verification techniques for neural network fairness. Our findings provide future research direction towards developing automated tools to detect bias in ML pipeline and instrument modules accordingly. Library developers can also provide API support for fairness control and monitoring using our methods. We conducted extensive experiments by collecting real-world datasets/model, and empirically validating theories. Furthermore, most of the recent works in the area try to test or mitigate bias. On top of that, we showed how to provide formal guarantees through reasoning and verification, so that the systems are safe by design. The goal is to empower developers to stay on top of the rapidly changing requirements of fairness.

7.1 Future Work

7.1.1 Fairness Reasoning in Modular ML Based Software

Many real life classification tasks are composed of multiple classifiers [57, 32]. Fairness composition of the systems where each components (or modules) has its own local fairness has not been studied. Two kinds of fairness composition are introduced by Dwork *et al.* **1) Functional composition:** In many classification settings, more than one classifier work independently on the same task and final prediction is made by accumulating the outcome of the modules. As an example, for college admission, generally students apply to a few institutions. Even if each college has a fair selection procedure, it would be desired to evaluate the fairness of “OR” of the colleges’ decisions, which implies that one student is accepted in at least one college. In another case, the acceptance without financial aid can be meaningless. Therefore, the system would be fair if the “AND” of the decisions (acceptance and financial aid) are fair. Here, we can study the fairness of independent classifiers (same task) and their composition through logical operators. **2) Dependent composition:** In another classification setting, the classifiers are applied in a sequence to reach the final outcome. For example, in hiring employees, a set of interviews lead to the final selection. Each classifier might filter candidates for the next classifier or output scores that are used later. Bower *et al.* showed that fair classifiers does not always lead to fair final classification [32]. We observed that other constraints (e.g., minimum filtering in a stage, subset of candidates available) also affect the fairness composition.

The research questions that can be asked in the above two situations are – 1) What are the compositional properties of fairness? Grgic *et al.* showed modular properties of fairness measures where modules are defined by subset of data features [81], i.e., whether the measures are supermodular, submodular or non-increasing monotone. The modular properties of the fairness measures would facilitate imposing constraints for fair software. 2) Does the existing metrics follow compositions? If not, how to measure them? The fairness metric EOD has been extended by Bower *et al.* to show that it follows the multiplicative composition [32]. We can analyze the other existing metrics and their modular properties. Early findings suggest that certain metrics produce local fairness signals which is violated in global fairness scenario. 3) How to ensure fairness in compositional systems? Understanding the modular measures of fairness, we can build the mechanism to improve the salient local outcomes, which affect meaningfully fair system. For example, Wang *et al.* proposed fairness mechanisms by leveraging the fairness composition of the modules in ranking problem of recommendation system [200].

7.1.2 Towards Fair and Interpretable Machine Learning Models

Several fairness metrics have been formed based on different definitions. The impossibility theory of fairness suggests that all the metrics can not be satisfied at the same time. Hence, it is necessary to document the requirements in advance as well as ensure that the decisions are explainable with respect to the given criteria. Only providing quantitative measure have little value to the developers who need to fix the issue.

Research also showed that developers often struggle to identify exact process of learning unfair representations or localize actual defects because of the black-box nature of ML. A future would be to define fairness as a first-class property in the learning process, and make decision-making interpretable. A more interpretable decision making would further enable accountability to the end-users given the complex fairness requirements. My previous research on verification and counterexample-driven repair shows potential for software interpretability and accountability.

7.1.3 Fairness Defect Repairing

Along with the accuracy and other functional issues, we need tools and methods to repair unfairness defect in ML-based software. Our works showed prospect for fairness defect repair in ML based software. For example, Fairify showed that we can do biased neuron identification and removal and counterexample-driven neural repair. Also, in our empirical study we outlined many fairness bugs that can repaired by devising multi-objective optimization.

Previous studies proposed bias mitigation algorithms that operate in specific situations and do not generalize to various datasets/models. Furthermore, those methods often lose accuracy while improving fairness. So, a future work can be to propose a novel approach to utilize the automated machine learning (AutoML) technique to mitigate bias. We can leverage AutoML techniques e.g., Bayesian multi-objective optimization in Auto-SkLearn, to improve fairness and accuracy together. We can improve upon the default optimization function of AutoML and incorporate fairness objectives with little to no loss of accuracy. We can propose methods to generate new optimization functions automatically based on an input dataset and targeted ML models. Another challenge with the existing AutoML techniques is its computational cost to find an optimum solution. An interesting future work would be to propose a fairness-aware search space pruning method for AutoML to significantly reduce the repair time.

BIBLIOGRAPHY

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

[2] Abdulla, W. (2017). Mask r-cnn for object detection and instance segmentation on keras and tensorflow. https://github.com/matterport/Mask_RCNN.

[3] Adebayo, J. and Kagal, L. (2016). Iterative orthogonal feature projection for diagnosing bias in black-box models. *arXiv preprint arXiv:1611.04967*.

[4] Adebayo, J. A. et al. (2016). *FairML: ToolBox for diagnosing bias in predictive modeling*. PhD thesis, Massachusetts Institute of Technology.

[5] Aggarwal, A., Lohia, P., Nagar, S., Dey, K., and Saha, D. (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*, pages 625–635.

[6] Albarghouthi, A., D'Antoni, L., Drews, S., and Nori, A. V. (2017). Fairsquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30.

[7] Alexis Chan, Octavio Arriaga, J. H. (2017). Autopilot-tensorflow. https://github.com/SullyChen/Autopilot-TensorFlow.

[8] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. (2019a). Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE.

[9] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. (2019b). Software engineering for machine learning: A case study. In *Proceedings of the 41st International Conference on Software Engineering*. ACM.

[10] Angwin, J., Larson, J., Mattu, S., and Kirchner, L. (2016). Machine bias: There's software used across the country to predict future criminals. *And it's biased against blacks. ProPublica*.

[11] Arriaga, O. (2018). Face classification and detectionn. https://github.com/oarriaga/face_classification.

[12] Ashmore, R., Calinescu, R., and Paterson, C. (2021). Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ACM Comput. Surv.*, 54(5).

[13] Athalye, A., Carlini, N., and Wagner, D. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International conference on machine learning*, pages 274–283. PMLR.

[14] Aungiers, J. (2019). Lstm neural network for time series prediction. https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction.

[15] Bantilan, N. (2018). Themis-ml: A fairness-aware machine learning interface for end-to-end discrimination discovery and mitigation. *Journal of Technology in Human Services*, 36(1):15–30.

[16] Barrett, C. and Tinelli, C. (2018). Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer.

[17] Bastani, O., Zhang, X., and Solar-Lezama, A. (2019). Probabilistic verification of fairness properties via concentration. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27.

[18] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. (2017). Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM.

[19] Bellamy, R. K., Dey, K., Hind, M., Hoffman, S. C., Houde, S., Kannan, K., Lohia, P., Martino, J., Mehta, S., Mojsilovic, A., et al. (2018). Ai fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *arXiv preprint arXiv:1810.01943*.

[20] Berk, R., Heidari, H., Jabbari, S., Kearns, M., and Roth, A. (2018). Fairness in criminal justice risk assessments: The state of the art. *Sociological Methods & Research*, page 0049124118782533.

[21] Berman, F., Rutenbar, R., Hailpern, B., Christensen, H., Davidson, S., Estrin, D., Franklin, M., Martonosi, M., Raghavan, P., Stodden, V., et al. (2018). Realizing the potential of data science. *Communications of the ACM*, 61(4):67–72.

[22] Binns, R. (2017). Fairness in machine learning: Lessons from political philosophy. *arXiv preprint arXiv:1712.03586*.

[23] Biswas, S. (2022). sumonbis/DS-Pipeline: Artifact for the ICSE 2022 Paper Entitled "The Art and Practice of Data Science Pipelines".

[24] Biswas, S., Islam, M. J., Huang, Y., and Rajan, H. (2019a). Boa meets python: A boa dataset of data science software in python language. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 577–581.

[25] Biswas, S., Islam, M. J., Huang, Y., and Rajan, H. (2019b). Boa meets python: a boa dataset of data science software in python language. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 577–581. IEEE Press.

[26] Biswas, S. and Rajan, H. Replication package for "fair preprocessing: Towards understanding compositional fairness of data transformers in machine learning pipeline".

[27] Biswas, S. and Rajan, H. (2020a). Do the machine learning models on a crowd sourced platform exhibit bias? an empirical study on model fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 642–653, New York, NY, USA. Association for Computing Machinery.

[28] Biswas, S. and Rajan, H. (2020b). ML-Fairness: Accepted Artifact for ESEC/FSE 2020 Paper on Fairness of Machine Learning Models. https://doi.org/10.5281/zenodo.3912064.

[29] Biswas, S. and Rajan, H. (2021a). 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*, ESEC/FSE 2021, page 981–993, New York, NY, USA. Association for Computing Machinery.

[30] Biswas, S. and Rajan, H. (2021b). Replication package for "fair preprocessing: Towards understanding compositional fairness of data transformers in machine learning pipeline".

[31] Biswas, S., Wardat, M., and Rajan, H. (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'22: The 44th International Conference on Software Engineering*.

[32] Bower, A., Kitchen, S. N., Niss, L., Strauss, M. J., Vargas, A., and Venkatasubramanian, S. (2017). Fair pipelines. *arXiv preprint arXiv:1707.00391*.

[33] Britz, D. (2018). Convolutional neural network for text classification in tensorflow. https://github.com/dennybritz/cnn-text-classification-tf.

[34] Brun, Y. and Meliou, A. (2018). Software fairness. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 754–759.

[35] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., et al. (2013). API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*.

[36] Calders, T. and Verwer, S. (2010). Three naive bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery*, 21(2):277–292.

[37] Carlini, N. and Wagner, D. (2017). Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE.

[38] Carpenter, J. (2011). May the best analyst win.

[39] Chakraborty, J., Majumder, S., and Menzies, T. (2021). Bias in machine learning software: Why? how? what to do? ESEC/FSE 2021, page 429–440, New York, NY, USA. Association for Computing Machinery.

[40] Chakraborty, J., Majumder, S., Yu, Z., and Menzies, T. (2020a). 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*, pages 654–665.

[41] Chakraborty, J., Peng, K., and Menzies, T. (2020b). Making fair ml software using trustworthy explanation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1229–1233. IEEE.

[42] Chakraborty, J., Xia, T., Fahid, F. M., and Menzies, T. (2019). Software engineering for fairness: A case study with hyperparameter optimization. *arXiv preprint arXiv:1905.05786*.

[43] Chandrasekar, P. and Qian, K. (2016). The impact of data preprocessing on the performance of a naive bayes classifier. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 618–619. IEEE.

[44] Chaudhuri, S., Gulwani, S., and Lublinerman, R. (2012). Continuity and robustness of programs. *Communications of the ACM*, 55(8):107–115.

[45] Chen, C. P. and Zhang, C.-Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347.

[46] Chen, J., Kallus, N., Mao, X., Svacha, G., and Udell, M. (2019). Fairness under unawareness: Assessing disparity when protected class is unobserved. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, pages 339–348.

[47] Chen Mengda, Z. M. (2018). reproduce mtcnn,a joint face detection and alignment using multi-task cascaded convolutional networks. https://github.com/AITTSMD/MTCNN-Tensorflow.

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

[49] Crone, S. F., Lessmann, S., and Stahlbock, R. (2006). The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing. *European Journal of Operational Research*, 173(3):781–800.

[50] Dahl, G. E., Jaitly, N., and Salakhutdinov, R. (2014). Multi-task neural networks for qsar predictions. *arXiv preprint arXiv:1406.1231*.

[51] D'Amour, A., Srinivasan, H., Atwood, J., Baljekar, P., Sculley, D., and Halpern, Y. (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*, pages 525–534.

[52] Dat Tran, S. C. (2018). Real-time object recognition app with tensorflow and opencv. https://github.com/datitran/object_detector_app.

[53] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[54] Dixon, L., Li, J., Sorensen, J., Thain, N., and Vasserman, L. (2018). Measuring and mitigating unintended bias in text classification. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 67–73.

[55] Dong, X., Chen, S., and Pan, S. J. (2017). Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 4860–4874.

[56] Dwork, C., Hardt, M., Pitassi, T., Reingold, O., and Zemel, R. (2012). Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*, pages 214–226.

[57] Dwork, C. and Ilvento, C. (2018). Fairness under composition. *arXiv preprint arXiv:1806.06122*.

[58] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431.

[59] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2015). Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34.

[60] Earl, M. (2016). Using neural networks to build an automatic number plate recognition system. https://github.com/matthewearl/deep-anpr.

[61] Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer.

[62] Elboher, Y. Y., Gottschlich, J., and Katz, G. (2020). An abstraction-based framework for neural network verification. In *International Conference on Computer Aided Verification*, pages 43–65. Springer.

[63] Engstrom, L., Ilyas, A., and Athalye, A. (2018). Evaluating and understanding the robustness of adversarial logit pairing. *arXiv preprint arXiv:1807.10272*.

[64] Feldman, M., Friedler, S. A., Moeller, J., Scheidegger, C., and Venkatasubramanian, S. (2015). Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 259–268.

[65] Friedler, S. A., Scheidegger, C., Venkatasubramanian, S., Choudhary, S., Hamilton, E. P., and Roth, D. (2019). A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, pages 329–338.

[66] Galhotra, S., Brun, Y., and Meliou, A. (2017). Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 498–510.

[67] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, page 406–431, Berlin, Heidelberg. Springer-Verlag.

[68] Gandomi, A. and Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International journal of information management*, 35(2):137–144.

[69] Garcia, R., Sreekanti, V., Yadwadkar, N., Crankshaw, D., Gonzalez, J. E., and Hellerstein, J. M. (2018). Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, volume 114.

[70] Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101.

[71] Garreau, D. and Luxburg, U. (2020). Explaining the explainer: A first theoretical analysis of lime. In *International Conference on Artificial Intelligence and Statistics*, pages 1287–1296. PMLR.

[72] Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. (2018). Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE.

[73] Goh, G., Cotter, A., Gupta, M., and Friedlander, M. P. (2016). Satisfying real-world goals with dataset constraints. In *Advances in Neural Information Processing Systems*, pages 2415–2423.

[74] Goodall, N. J. (2016). Can you program ethics into a self-driving car? *IEEE Spectrum*, 53(6):28–58.

[75] Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

[76] Google Cloud Blog (2019). Machine learning workflow. `https://cloud.google.com/ml-engine/docs/tensorflow/ml-solutions-overview`.

[77] Gopinath, D., Converse, H., Pasareanu, C., and Taly, A. (2019a). Property inference for deep neural networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 797–809. IEEE.

[78] Gopinath, D., Katz, G., Păsăreanu, C. S., and Barrett, C. (2018a). Deepsafe: A data-driven approach for assessing robustness of neural networks. In *International symposium on automated technology for verification and analysis*, pages 3–19. Springer.

[79] Gopinath, D., Wang, K., Zhang, M., Pasareanu, C. S., and Khurshid, S. (2018b). Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439*.

[80] Gopinath, D., Zhang, M., Wang, K., Kadron, I. B., Pasareanu, C., and Khurshid, S. (2019b). Symbolic execution for importance analysis and adversarial generation in neural networks. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 313–322. IEEE.

[81] Grgic-Hlaca, N., Zafar, M. B., Gummadi, K. P., and Weller, A. (2018). Beyond distributive fairness in algorithmic decision making: Feature selection for procedurally fair learning. In *AAAI*, pages 51–60.

[82] Guo, Y., Yao, A., and Chen, Y. (2016). Dynamic network surgery for efficient dnns. *arXiv preprint arXiv:1608.04493*.

[83] Han, S., Pool, J., Tran, J., and Dally, W. (2015). Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems*, 28.

[84] Hardt, M., Price, E., and Srebro, N. (2016). Equality of opportunity in supervised learning. In *Advances in neural information processing systems*, pages 3315–3323.

[85] Harrison, G., Hanson, J., Jacinto, C., Ramirez, J., and Ur, B. (2020). An empirical study on the perceived fairness of realistic, imperfect machine learning models. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, pages 392–402.

[86] Hill, C., Bellamy, R., Erickson, T., and Burnett, M. (2016). Trials and tribulations of developers of intelligent systems: A field study. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 162–170. IEEE.

[87] Hofmann, D. H. (1994). German credit dataset: UCI machine learning repository.

[88] Holstein, K., Wortman Vaughan, J., Daumé III, H., Dudik, M., and Wallach, H. (2019). Improving fairness in machine learning systems: What do industry practitioners need? In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–16.

[89] Hong, S. A. and Hunter, T. (2017). Build, scale, and deploy deep learning pipelines with ease. https://databricks.com/blog/2017/09/06/build-scale-deploy-deep-learning-pipelines-ease.html.

[90] Hort, M., Zhang, J. M., Sarro, F., and Harman, M. (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 (to appear)*.

[91] Huang, R., Xu, B., Schuurmans, D., and Szepesvári, C. (2015). Learning with a strong adversary. *arXiv preprint arXiv:1511.03034*.

[92] Huang, X., Kwiatkowska, M., Wang, S., and Wu, M. (2017). Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer.

[93] Huchard, M., Kästner, C., and Fraser, G. (2018). Proceedings of the 33rd acm/ieee international conference on automated software engineering (ase 2018). In *ASE: Automated Software Engineering*. ACM Press.

[94] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. (2019). A comprehensive study on deep learning bug characteristics. In *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[95] Islam, M. J., Pan, R., Nguyen, G., and Rajan, H. (2020). Repairing deep neural networks: Fix patterns and challenges. In *ICSE'20: The 42nd International Conference on Software Engineering*.

[96] Jepsen, K. (2014). The machine learning community takes on the higgs. https://www.symmetrymagazine.org/article/july-2014/the-machine-learning-community-takes-on-the-higgs.

[97] John, P. G., Vijaykeerthy, D., and Saha, D. (2020). Verifying individual fairness in machine learning models. In *Conference on Uncertainty in Artificial Intelligence*, pages 749–758. PMLR.

[98] Joshi, P. (2017). *Artificial intelligence with python*. Packt Publishing Ltd.

[99] Kaggle (2010). The world's largest data science community with powerful tools and resources to help you achieve your data science goals. www.kaggle.com.

[100] Kaggle (2017a). Adult Census Dataset. https://www.kaggle.com/uciml/adult-census-income.

[101] Kaggle (2017b). Bank Marketing Dataset. https://www.kaggle.com/c/bank-marketing-uci.

[102] Kaggle (2017c). Competition: Santander Product Recommendation. https://www.kaggle.com/c/santander-product-recommendation/overview.

[103] Kaggle (2017d). German Credit Dataset. https://www.kaggle.com/uciml/german-credit.

[104] Kaggle (2017e). Home Credit Dataset. https://www.kaggle.com/c/home-credit-default-risk.

[105] Kaggle (2017f). Titanic ML Dataset. https://www.kaggle.com/c/titanic/data.

[106] Kaggle (2017g). Titanic ML Dataset. https://www.kaggle.com/c/titanic/data.

[107] Kaggle (2019a). Adult Census Kernel: Multiple ML Techniques and Analysis. https://www.kaggle.com/bananuhbeatdown/multiple-ml-techniques-and-analysis-of-dataset.

[108] Kaggle (2019b). Kernel: German Credit Risk Analysis. https://www.kaggle.com/pahulpreet/german-credit-risk-analysis-beginner-s-guide.

[109] Kaggle (2021a). Kaggle Notebook. www.kaggle.com/thousandvoices/simple-lstm.

[110] Kaggle (2021b). Kaggle Notebook. https://www.kaggle.com/zfturbo/simple-ru-baseline-lb-0-9627.

[111] Kaggle (2021c). Kaggle Notebook. www.kaggle.com/seesee/siamese-pretrained-0-822.

[112] Kaggle (2021d). Kaggle Notebook. www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction.

[113] Kamiran, F. and Calders, T. (2012). Data preprocessing techniques for classification without discrimination. *Knowledge and Information Systems*, 33(1):1–33.

[114] Kamiran, F., Karim, A., and Zhang, X. (2012). Decision theory for discrimination-aware classification. In *2012 IEEE 12th International Conference on Data Mining*, pages 924–929. IEEE.

[115] Kamishima, T., Akaho, S., Asoh, H., and Sakuma, J. (2012). Fairness-aware classifier with prejudice remover regularizer. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 35–50. Springer.

[116] Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017a). Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer.

[117] Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017b). Towards proving the adversarial robustness of deep neural networks. *arXiv preprint arXiv:1709.02802*.

[118] Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al. (2019). The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer.

[119] Keras (2021a). Keras API Reference. https://keras.io/api/.

[120] Keras (2021b). Scikit-Learn API Reference. https://scikit-learn.org/stable/modules/classes.html.

[121] Kim, M., Zimmermann, T., DeLine, R., and Begel, A. (2016). The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, pages 96–107. ACM.

[122] Kim, N. (2018). Speech-to-text-wavenet : End-to-end sentence level english speech recognition based on deepmind's wavenet and tensorflow. https://github.com/buriburisuri/speech-to-text-wavenet.

[123] Kirkpatrick, K. (2017). It's not the algorithm, it's the data. *Communications of the ACM*, 60(2):21–23.

[124] Kleinberg, J., Mullainathan, S., and Raghavan, M. (2016). Inherent trade-offs in the determination of risk scores. *arXiv preprint arXiv:1609.05807*.

[125] Kohavi, R. (1996). Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *KDD*, volume 96, pages 202–207.

[126] Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T., and Markl, V. (2019). An intermediate representation for optimizing machine learning pipelines. *Proceedings of the VLDB Endowment*, 12(11):1553–1567.

[127] Kusner, M., Loftus, J., Russell, C., and Silva, R. (2017). Counterfactual fairness. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 4069–4079.

[128] Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563.

[129] Lientz, B. P., Swanson, E. B., and Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471.

[130] Ma, S., Liu, Y., Lee, W.-C., Zhang, X., and Grama, A. (2018). Mode: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 175–186.

[131] Mazzucato, D. and Urban, C. (2021). Reduced products of abstract domains for fairness certification of neural networks. In *International Static Analysis Symposium*, pages 308–322. Springer.

[132] Miao, H., Li, A., Davis, L. S., and Deshpande, A. (2017). Towards unified data and lifecycle management for deep learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 571–582. IEEE.

[133] Moro, S., Cortez, P., and Rita, P. (2014). A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62:22–31.

[134] Nguyen, A., Yosinski, J., and Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 427–436.

[135] Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., García, Á. L., Heredia, I., Malík, P., and Hluchỳ, L. (2019). Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, pages 1–48.

[136] Olson, P. (2011). The algorithm that beats your bank manager. *CNN Money*.

[137] Olson, R. S., Bartley, N., Urbanowicz, R. J., and Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 485–492. ACM.

[138] Olson, R. S. and Moore, J. H. (2016). Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR.

[139] Paino, A. (2017). Deep learning models trained to correct input errors in short, message-like text. https://github.com/atpaino/deep-text-corrector.

[140] Pan, R. and Rajan, H. (2020). On decomposing a deep neural network into modules. In *ESEC/FSE'2020: The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[141] Park, K. (2018). A tensorflow implementation of tacotron: A fully end-to-end text-to-speech synthesis model. https://github.com/Kyubyong/tacotron.

[142] Parnas, D. L., Clements, P. C., and Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on software Engineering*, (3):259–266.

[143] Paulsen, B., Wang, J., and Wang, C. (2020). Reludiff: Differential verification of deep neural networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 714–726. IEEE.

[144] Pearl, J. (2000). Causality: Models, reasoning and inference cambridge university press. *Cambridge, MA, USA,*, 9:10–11.

[145] Pearl, J. et al. (2009). Causal inference in statistics: An overview. *Statistics surveys*, 3:96–146.

[146] Pedreshi, D., Ruggieri, S., and Turini, F. (2008). Discrimination-aware data mining. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 560–568.

[147] Pei, K., Cao, Y., Yang, J., and Jana, S. (2017). Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18.

[148] Pleiss, G., Raghavan, M., Wu, F., Kleinberg, J., and Weinberger, K. Q. (2017). On fairness and calibration. In *Advances in Neural Information Processing Systems*, pages 5680–5689.

[149] Polyzotis, N., Roy, S., Whang, S. E., and Zinkevich, M. (2017). Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1723–1726. ACM.

[150] Polyzotis, N., Roy, S., Whang, S. E., and Zinkevich, M. (2018). Data lifecycle challenges in production machine learning: A survey. *ACM SIGMOD Record*, 47(2):17–28.

[151] Pulina, L. and Tacchella, A. (2010). An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, pages 243–257. Springer.

[152] Qi, C. (2019). Caffe implementation of google mobilenet ssd detection network. https://github.com/chuanqi305/MobileNet-SSD.

[153] Rajlich, V. (2014). Software evolution and maintenance. In *Future of Software Engineering Proceedings*, pages 133–144.

[154] Rehman, M. H., Chang, V., Batool, A., and Wah, T. Y. (2016). Big data reduction framework for value creation in sustainable enterprises. *International Journal of Information Management*, 36(6):917–928.

[155] Ribeiro, M. T., Singh, S., and Guestrin, C. (2018). Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

[156] Roh, Y., Heo, G., and Whang, S. E. (2019). A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*.

[157] Ruan, E. (2019). Scene text detection based on ctpn (connectionist text proposal network). https://github.com/eragonruan/text-detection-ctpn.

[158] Rule, A., Tabard, A., and Hollan, J. D. (2018). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

[159] Ruoss, A., Balunovic, M., Fischer, M., and Vechev, M. (2020). Learning certified individually fair representations. *Advances in Neural Information Processing Systems 33 pre-proceedings (NeurIPS 2020)*.

[160] Russell, C., Kusner, M. J., Loftus, J. R., and Silva, R. (2017). When worlds collide: integrating different counterfactual assumptions in fairness. *Advances in Neural Information Processing Systems 30. Pre-proceedings*, 30.

[161] Rémy, P. (2018). Deep learning model to analyze a large corpus of clear text passwords. https://github.com/philipperemy/tensorflow-1.4-billion-password-analysis.

[162] Saleiro, P., Kuester, B., Hinkson, L., London, J., Stevens, A., Anisfeld, A., Rodolfa, K. T., and Ghani, R. (2018). Aequitas: A bias and fairness audit toolkit. *arXiv preprint arXiv:1811.05577*.

[163] Salimi, B., Rodriguez, L., Howe, B., and Suciu, D. (2019). Interventional fairness: Causal database repair for algorithmic fairness. In *Proceedings of the 2019 International Conference on Management of Data*, pages 793–810.

[164] Sandberg, D. (2018). Face recognition using tensorflow. https://github.com/davidsandberg/facenet.

[165] Sapp, C. E. (2017). Preparing and architecting machine learning. *Gartner Technical Professional Advice*, pages 1–37.

[166] Scikit Learn (2019a). Feature Selection Methods. https://scikit-learn.org/stable/modules/feature_selection.html.

[167] Scikit Learn (2019b). LightGBM API Documentation. https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html.

[168] Scikit Learn (2019c). Preprocessing API Documentation. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing.

[169] Scikit Learn (2019d). Scikit Learn SimpleImputer. https://scikit-learn.org/0.18/modules/generated/sklearn.preprocessing.Imputer.html.

[170] Scikit Learn (2019e). SVC API Documentation. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[171] Scikit-Learn Pipeline (2020). Scikit-Learn API Documentation. https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html.

[172] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., and Young, M. (2014). Machine learning: The high interest credit card of technical debt.

[173] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511.

[174] Severtson, R. B. (2017). What is the team data science process? https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview.

[175] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., USA.

[176] Shriver, D., Elbaum, S., and Dwyer, M. B. (2021a). Dnnv: A framework for deep neural network verification. In *International Conference on Computer Aided Verification*, pages 137–150. Springer.

[177] Shriver, D., Elbaum, S., and Dwyer, M. B. (2021b). DNNV Documentation. https://docs.dnnv.org/en/stable/dnnp/introduction.html.

[178] Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2019). An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30.

[179] Sokol, K., Santos-Rodriguez, R., and Flach, P. (2019). Fat forensics: A python toolbox for algorithmic fairness, accountability and transparency. *arXiv preprint arXiv:1909.05167*.

[180] Song, G. (2017). Tensorflow-based recommendation systems. https://github.com/songgc/TF-recomm.

[181] Speicher, T., Heidari, H., Grgic-Hlaca, N., Gummadi, K. P., Singla, A., Weller, A., and Zafar, M. B. (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*, pages 2239–2248.

[182] Stack Overflow (2016). How does the class_weight parameter in scikit-learn work? https://stackoverflow.com/questions/30972029/how-does-the-class-weight-parameter-in-scikit-learn-work.

[183] Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., and Kroening, D. (2018). Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 109–119.

[184] Szegedy, C. (2014). Google inc, wojciech zaremba, ilya sutskever, google inc, joan bruna, dumitru erhan, google inc, ian goodfellow, and rob fergus. intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*.

[185] Teichmann, M. (2018). A kitti road segmentation model implemented in tensorflow. https://github.com/MarvinTeichmann/KittiSeg.

[186] Tian, Y., Pei, K., Jana, S., and Ray, B. (2018). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314.

[187] Todd, S. and Dietrich, D. (2017). Computing resource re-provisioning during data analytic lifecycle. US Patent 9,619,550.

[188] Tramer, F., Atlidakis, V., Geambasu, R., Hsu, D., Hubaux, J.-P., Humbert, M., Juels, A., and Lin, H. (2017). Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 401–416. IEEE.

[189] Trieu, A. B. (2018). Real-time object detection and classification. https://github.com/thtrieu/darkflow.

[190] Udeshi, S., Arora, P., and Chattopadhyay, S. (2018). Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 98–108.

[191] Urban, C., Christakis, M., Wüstholz, V., and Zhang, F. (2020). Perfectly parallel fairness certification of neural networks. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30.

[192] US Equal Employment Opportunity Commission (1979). Guidelines on Employee Selection Procedures.

[193] Uysal, A. K. and Gunal, S. (2014). The impact of preprocessing on text classification. *Information Processing & Management*, 50(1):104–112.

[194] Van Der Weide, T., Papadopoulos, D., Smirnov, O., Zielinski, M., and Van Kasteren, T. (2017). Versioning for end-to-end machine learning pipelines. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning*, page 2. ACM.

[195] Viera, A. J., Garrett, J. M., et al. (2005). Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363.

[196] Wagner, B. (2020). Accountability by design in technology research. *Computer Law & Security Review*, 37:105398.

[197] Wan, Z., Xia, X., Lo, D., and Murphy, G. C. (2019). How does machine learning change software development practices? *IEEE Transactions on Software Engineering*.

[198] Wang, J., Li, L., and Zeller, A. (2021). Restoring execution environments of jupyter notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1622–1633. IEEE.

[199] Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018). Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems*, 31.

[200] Wang, X., Thain, N., Sinha, A., Chi, E. H., Chen, J., and Beutel, A. (2019). Practical compositional fairness: Understanding fairness in multi-task ml systems. *arXiv preprint arXiv:1911.01916*.

[201] Wardat, M., Le, W., and Rajan, H. (2021). Deeplocalize: Fault localization for deep neural networks. In *ICSE'21: The 43nd International Conference on Software Engineering*.

[202] Wickham, H. (2019). Data science: how is it different to statistics? *IMS Bulletin*, 48.

[203] Wing, J. M. (2019). The data life cycle. *Harvard Data Science Review*.

[204] Wirth, R. and Hipp, J. (2000). Crisp-dm: Towards a standard process model for data mining. In *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, pages 29–39. Citeseer.

[205] Woolf, M. (2018). Automatically "block" people in images (like black mirror) using a pretrained neural network. https://github.com/minimaxir/person-blocker.

[206] Xiao, Y., Beschastnikh, I., Rosenblum, D. S., Sun, C., Elbaum, S., Lin, Y., and Dong, J. S. (2021). Self-checking deep neural networks in deployment. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 372–384. IEEE.

[207] Yang, K., Huang, B., Stoyanovich, J., and Schelter, S. (2020). Fairness-aware instrumentation of preprocessing pipelines for machine learning. In *Workshop on Human-In-the-Loop Data Analytics (HILDA'20)*.

[208] Yu, A. W., Dohan, D., Luong, M.-T., Zhao, R., Chen, K., Norouzi, M., and Le, Q. V. (2018). A tensorflow implementation of qanet for machine reading comprehension. https://github.com/NLPLearn/QANet.

[209] Yurochkin, M., Bower, A., and Sun, Y. (2019). Training individually fair ml models with sensitive subspace robustness. In *International Conference on Learning Representations*.

[210] Zafar, M. B., Valera, I., Rodriguez, M. G., and Gummadi, K. P. (2015). Fairness constraints: Mechanisms for fair classification. *arXiv preprint arXiv:1507.05259*.

[211] Zelaya, C. V. G. (2019). Towards explaining the effects of data preprocessing on machine learning. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2086–2090. IEEE.

[212] Zemel, R., Wu, Y., Swersky, K., Pitassi, T., and Dwork, C. (2013). Learning fair representations. In *International Conference on Machine Learning*, pages 325–333.

[213] Zhang, B. H., Lemoine, B., and Mitchell, M. (2018). Mitigating unwanted biases with adversarial learning. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 335–340.

[214] Zhang, F., Chowdhury, S. P., and Christakis, M. (2020a). Deepsearch: A simple and effective blackbox attack for deep neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 800–812.

[215] Zhang, J. and Bareinboim, E. (2018). Fairness in decision-making—the causal explanation formula. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

[216] Zhang, J. M. and Harman, M. (2021). "ignorance and prejudice" in software fairness. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1436–1447. IEEE.

[217] Zhang, P., Wang, J., Sun, J., Dong, G., Wang, X., Wang, X., Dong, J. S., and Dai, T. (2020b). White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 949–960.

[218] Zheng, H., Chen, Z., Du, T., Zhang, X., Cheng, Y., Ji, S., Wang, J., Yu, Y., and Chen, J. (2022). Neuronfair: Interpretable white-box fairness testing through biased neuron identification.

[219] Zhilin Yang, Zihang Dai, C. B. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. https://github.com/zihangdai/xlnet.

[220] Zhou, L. (2019). How to build a better machine learning pipeline. https://www.datanami.com/2018/09/05/how-to-build-a-better-machine-learning-pipeline.