Fairness as a Program Property

Aws Albarghouthi, Loris D'Antoni, Samuel Drews
University of Wisconsin–Madison

Aditya Nori
Microsoft Research

Abstract We explore the following question: *Is a decision-making program fair, for some useful definition of fairness?* First, we describe how several *algorithmic fairness* questions can be phrased as *program verification problems*. Second, we discuss an automated verification technique for proving or disproving fairness of decision-making programs with respect to a *model of the population*.

1. Introduction

Algorithms have become powerful arbitrators of a range of significant decisions with far-reaching societal impact—hiring [21, 22], welfare allocation [15], prison sentencing [2], policing [5, 25], amongst many others. With the range and sensitivity of algorithmic decisions expanding by the day, the question of whether an algorithm is *fair* is a pressing one. Indeed, the notion of *algorithmic fairness* has captured the attention of a broad spectrum of experts: machine learning and theory researchers [6, 13, 16, 29]; privacy researchers and investigative journalists [2, 10, 26, 28]; law scholars and social scientists [1, 3, 27]; governmental agencies and NGOs [24].

Ultimately, algorithmic fairness is a question about programs and their properties: *Is a given program $\mathcal{P}$ fair, under some definition of fairness? Or, how fair is $\mathcal{P}$?* In this paper, we describe a line of work that approaches the question of algorithmic fairness from a program-analytic perspective, in which our goal is to *analyze* a given decision-making program and *construct a proof* of its fairness or unfairness—just as a traditional static program verifier would prove correctness of a program with respect to, for example, lack of divisions by zero, integer overflows, null-pointer derefrences, etc.

We start by analyzing what are the challenges and research questions in checking algorithmic fairness for decision making programs (Section 2). We then present a simple case study and show how techniques for verifying probabilistic programs can be used to automatically prove or disprove global fairness for a class of programs that subsume a range of machine learning classifiers (Section 3). Finally, we lay a list of many challenging and interesting questions that the algorithms and programming languages communities need to answer to achieve the ultimate goal of building a fully automated system for verifying and guaranteeing algorithmic fairness in real-world applications (Section 4).

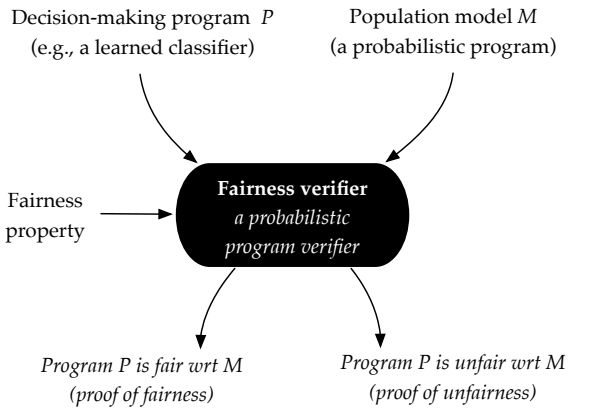

Figure 1. Overview

2. Proving Programs Fair

In this section, we describe the components of the fairness verification problem. Intuitively, our goal is to prove whether a certain program is fair with respect to the set of possible inputs over which it operates. Tackling the fairness-verification problem requires answering a number of challenging questions:

– What class of decision-making programs should our program model capture?

– How can we define the set of possible inputs to the program and capture complex probability distributions that are useful and amenable to verification?
– How can we describe what it means for the program to be fair?
– How can we fully automate the verification process?

Figure 1 provides a high-level picture of our proposed framework. As shown, the *fairness verifier* takes a (white-box) decision-making program $\mathcal{P}$ and a population model M. The verifier then proceeds to prove or disprove that $\mathcal{P}$ is fair for the given population defined by the model M. Here, the model M defines a joint probability distribution on the inputs of $\mathcal{P}$. Existing definitions of fairness define programs as fair or unfair with respect to a given concrete dataset. While using a concrete dataset simplifies the verification problem, it also raises questions of whether the dataset is representative for the population for which we are trying to prove fairness. Our technique moves away from concrete datasets and replaces them with a probabilistic population model. We envision a future in which fairness verification is regulated.[1] For instance, a governmental agency can publish a probabilistic population model (e.g., generated from census data). Any organization employing a decision-making algorithm with potentially significant consequences (e.g., hiring) must quantify fairness of their algorithmic process against the current picture of the population, as specified by the population model.

Decision-making programs In the context of algorithmic fairness, a program $\mathcal{P}$ takes as input a vector of arguments $\boldsymbol{v}$ representing a set of input attributes (features), where one (or more) of the arguments v_s in the vector $\boldsymbol{v}$ is *sensitive*—e.g., gender or race. Evaluating $\mathcal{P}(\boldsymbol{v})$ may return a Boolean value indicating—e.g., hire or not hire—if the program is a binary or a numerical value—e.g., a mortgage rate. The set of combinators, operations, and types used by the program can vastly affect the complexity of the verification procedures. For example, loops are the hardest type of programming construct to reason about, but most machine learning classifiers do not contain loops. Similarly, since classifiers typically operate over real values, we can limit the set of possible types allowed in our programs to only being reals or other types that can be *desugared* into reals. All these decisions are crucial in the design of a verification procedure.

Population model To be able to reason about the outcome of the program we need to specify what kind of input the program will operate on. For example, although a program that allocates mortgages might be "fair" with respect a certain set of applicants, it may become unfair when considering a different pool of people. In program verification, the "kind of inputs" over which the program operates is called the precondition and is typically stated as a formal logical property with the program inputs as free variables. An example of program precondition is

$$v_{gender} = f \rightarrow v_{job} \neq \textit{priest}$$

which indicates that none of the program inputs is both a woman and a priest. Of course, there are many possible choices for what language we can use to describe the program's precondition. In particular, if we want to capture a certain probability distribution over the input of the program, our language will be a logic that can describe probabilities and random variables. For example, we might want to be able to specify that half of the inputs are female, $\Pr[v_{gender} = f] = 0.5$, or that the age of the processed inputs has a particular distribution, $v_{age} \sim gauss(18, 5)$. Again, the choice of the language allowed in the preconditions is crucial in the design of a verification procedure. From now on, we refer to the program precondition, D_{pop}, as the *population model*.

Fairness properties The next step is to define a property stating that the program's outcome is fair with respect to the program's precondition. In program verification, this is called the postcondition of the program. As observed in the fairness literature, there are many ways to define when and why a program is fair or unfair.

For example, if we want to prove *group fairness*—i.e., that the algorithm is *just as likely* to hire a minority applicant (m) as it is for other, non-minority applicants—our postcondition will be an expression of the form

$$\frac{\Pr[\mathcal{P}(\boldsymbol{v}) = \textit{true} \mid v_s = m]}{\Pr[\mathcal{P}(\boldsymbol{v}) = \textit{true} \mid v_s \neq m]} > 1 - \epsilon$$

where *true* is the desired return value of the program, e.g., indicating hiring. On the other hand, if we want to prove *individual fairness*—i.e., *similar* inputs should

[1] The European Union (EU), for instance, has already begun regulating algorithmic decision-making [17].

have similar outcomes—our postcondition will be an expression of the form

$$\Pr[\mathcal{P}(\boldsymbol{v}) \neq \mathcal{P}(\boldsymbol{v'}) \mid \boldsymbol{v} \sim \boldsymbol{v'}] < \epsilon$$

Notice that the last postcondition relates the outcomes of the program on different input values. As the two types of properties we described are radically different, they will also require different verification mechanisms.

Proofs of (un)fairness The task of proving whether a program is fair boils down to statically checking whether, on inputs satisfying the precondition, the outcome of the program satisfies the post-condition. For simple definitions, such as group fairness, the verification problem reduces to computing the probability of a number of events with respect to the program and the population model. For more complex definitions, such as individual fairness, proving fairness requires more complex reasoning involving multiple runs of the programs (i.e., a *hyperproperty* [9]), a notoriously hard problem. In the case of a negative result, the verifier should provide the users with a proof of unfairness. Depending on the fairness definition, producing a human-readable proof might be challenging as the argument might involve multiple and potentially infinite inputs. For example, in the case of group fairness it might be challenging to explain why the program outputs true on 40% of the minority inputs and on 70% of the majority inputs.

3. Case Study

We now describe a simplified case study demonstrating how our fairness verification methodology can be used to prove or disprove fairness of a given decision-making program.

A program and a population model Consider the following program `dec`, which is a decision-making program that takes a job applicant's college ranking and years of experience and decides whether they get hired or not (the *fairness target*). The program implements a decision tree, perhaps one generated by a machine-learning algorithm. A person is hired if they attended a *top-5* college (`colRank <= 5`) or have lots of experience compared to their college's ranking (`expRank > -5`). Observe that `dec` *does not access ethnicity*.

```
define dec(colRank, yExp)
  expRank ← yExp - colRank
  if (colRank <= 5)
    hire ← true
  elif (expRank > -5)
    hire ← true
  else
    hire ← false
  return hire
```

Now, consider the program `popModel`, which is a probabilistic program describing a simple model of the population. Here, a member of the population has three attributes, all of which are real-valued: (*i*) `ethnicity`; (*ii*) `colRank`, the ranking of the college the person attended (lower is better); and (*iii*) `yExp`, the years of work experience a person has. We consider a person is a member of a protected group if `ethnicity > 10`; we call this the *sensitive condition*. The population model can be viewed as a *generative model* of records of individuals—the more likely a combination is to occur in the population, the more likely it will be generated. For instance, the years of experience an individual has (line 4) follows a Gaussian distribution with mean 10 and standard deviation 5.

```
define popModel()
  ethnicity ~ gauss(0,10)
  colRank ~ gauss(25,10)
  yExp ~ gauss(10,5)
  if (ethnicity > 10)
    colRank ← colRank + 5
  return colRank, yExp
```

A note on the program model Note that our program model, while admitting arbitrary programs, is rich enough to capture programs (classifiers) generated by standard machine learning algorithms. For example, linear support vector machines, decision trees, and neural networks, can be represented in our language simply using assignments with arithmetic expressions and conditionals. Similarly, the population model is a probabilistic program, where assignments can be made by drawing values from predefined distributions. Like other probabilistic programming languages, our programming model is rich enough to subsume graphical models like Bayesian networks [19].

Group fairness Suppose that our goal is to prove group fairness, following the definition of Feldman et al. [16]:

$$\frac{\Pr[\texttt{hire} \mid \texttt{min}]}{\Pr[\texttt{hire} \mid \neg\texttt{min}]} > 1 - \epsilon$$

where `min` is shorthand for the sensitive condition `ethnicity > 10`.

Probabilistic inference as volume computation To prove (un)fairness of the decision-making model with respect to the population, we need to compute the probabilities appearing in the group fairness ratio. For illustration, suppose we are computing the probability $\Pr[\texttt{hire} \wedge \neg\texttt{min}]$. We need to reason about the *composition* of the two programs, `dec` $\circ$ `popModel`. That is, we want to compute the probability that (*i*) `popModel` generates a non-minority applicant, and (*ii*) `dec` hires that applicant. To do so, we observe that every possible execution of the composition `dec` $\circ$ `popModel` is uniquely characterized by the set of the three probabilistic choices made by `popModel`. In other words, every execution is characterized by a vector $\boldsymbol{v} \in \mathbb{R}^3$.

Thus, our goal is to compute the probability that we draw a vector $\boldsymbol{v}$ that results in a minority applicant being hired. Probabilistic programming languages, e.g., Church [18], R2 [23], and Stan [7], employ *approximate inference* techniques, like MCMC, which converge in the limit but offer no guarantees on how far we are from the exact result. In our work, we consider *exact inference*, which has primarily received attention in the Bayesian network setting, and boils down to solving a #SAT instance [8]. In our setting, however, we are dealing with real-valued variables.

Using standard techniques from program analysis and verification, we can characterize the set of all such vectors as a formula φ, which is comprised of Boolean combinations (conjunctions/disjunctions) of linear inequalities—since our program only has linear expressions. Geometrically, the formula φ is a set of convex polyhedra in $\mathbb{R}^n$. Therefore, the probability $\Pr[\texttt{hire} \wedge \neg\texttt{min}]$ is the same as the probability of drawing a vector $\boldsymbol{v}$ that lies inside of φ. In other words, we are interested in the *volume* of φ, weighted by the probabilistic choices. Formally:

$$\Pr[\texttt{hire} \wedge \neg\texttt{min}] = \int_{\varphi} p_e p_y p_c \, dedydc$$

where, e.g., p_e is the *probability density function* of the distribution `gauss(0,10)`—the distribution from which the value of `ethnicity` is drawn in line 2 of `popModel`.

The volume computation problem is a well-studied and hard problem [14, 20]. Indeed, even for a convex polytope, computing its volume is #P-hard. Leveraging the great developments in *satisfiabiltiy modulo theories* (SMT) solvers [4], we developed a procedure that reduces the volume compuation problem to a series of

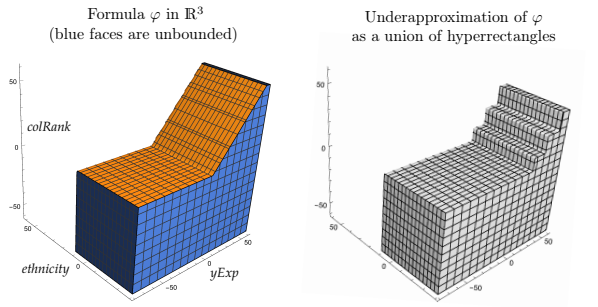

Figure 2. Underapproximation of φ as hyperrectangles

calls to the SMT solver, viewed completely as an oracle. Specifically, our procedure uses the SMT solver to *sample* subregions of φ that are *hyperrectangular*. Intuitively, for hyperrectangular regions in $\mathbb{R}^n$, evaluating the above integral is a matter of evaluating the CDFs of the various distributions. Thus, by systematically sampling more and more non-overlapping hyperrectangles in φ, we maintain a *lower bound* on the probability of interest. Figure 2 pictorially illustrates φ and an under-approximation with 4 hyperrectangles. Similarly, to compute an *upper bound* on the probability, we can simply invoke our procedure on $\neg\varphi$.

Fairness certificates The fairness verification tool terminates when it has computed lower/upper bounds that prove or disprove the desired fairness criteria. The hyperrectangles sampled in the process of computing volumes can serve as proof certificates. That is, an external entity can take the hyperrectangles, compute their volumes, and ensure that they indeed lie in the expected regions in $\mathbb{R}^n$.

4. Experience and future Outlook

Experience We have built a fairness-verification tool, called FairSquare, that takes a decision-making program, a population model, and verifies fairness of the program with respect to the model. So far, we have focused on group fairness. The tool uses the popular Z3 SMT solver [12] for manipulating first-order formulas over arithmetic theories.

We have used FairSquare to prove or disprove fairness of a suite of population models and programs representing machine-learning classifiers that were automatically generated from real-world datasets used in other work on algorithmic fairness [11, 16, 29]. Specifically, we have considered linear SVMs, simple neural networks with rectified linear units, and decision trees.

Future outlook Looking forward, we see a wide range of avenues for improvement and exploration. For in-

stance, we are currently working on the problem of *making an unfair program fair*. That is, given a program $\mathcal{P}$ that is considered unfair, what is the smallest *tweak* that would make it fair. Our goal is to *repair* the program, making it fair, while ensuring that it is semantically close to the original program.

References

[1] Ifeoma Ajunwa, Sorelle Friedler, Carlos E Scheidegger, and Suresh Venkatasubramanian. Hiring by algorithm: predicting and preventing disparate impact. *Available at SSRN 2746078*, 2016.

[2] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. Machine bias: There's software used across the country to predict future criminals. and it's biased against blacks. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing, May 2016. (Accessed on 06/18/2016).

[3] Solon Barocas and Andrew D Selbst. Big data's disparate impact. *Available at SSRN 2477899*, 2014.

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[5] Nate Berg. Predicting crime, lapd-style. https://www.theguardian.com/cities/2014/jun/25/predicting-crime-lapd-los-angeles-police-data-analysis-algorithm-minority-report, June 2014. (Accessed on 06/18/2016).

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

[7] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: a probabilistic programming language. *Journal of Statistical Software*, 2015.

[8] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008.

[9] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[10] Amit Datta, Michael Carl Tschantz, and Anupam Datta. Automated experiments on ad privacy settings. *Proceedings on Privacy Enhancing Technologies*, 2015(1):92–112, 2015.

[11] Anupam Datta, Shayak Sen, and Yair Zick. Algorithmic transparency via quantitative input influence. In *Proceedings of 37th IEEE Symposium on Security and Privacy*, 2016.

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

[13] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. Fairness through awareness. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 214–226, 2012.

[14] Martin E. Dyer and Alan M. Frieze. On the complexity of computing the volume of a polyhedron. *SIAM Journal on Computing*, 17(5):967–974, 1988.

[15] Virginia Eubanks. The dangers of letting algorithms enforce policy. http://www.slate.com/articles/technology/future_tense/2015/04/the_dangers_of_letting_algorithms_enforce_policy.html, April 2015. (Accessed on 06/18/2016).

[16] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 259–268, 2015.

[17] B. Goodman and S. Flaxman. EU regulations on algorithmic decision-making and a "right to explanation". *ArXiv e-prints*, June 2016.

[18] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[19] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

[20] Leonid Khachiyan. *Complexity of polytope volume computation*. Springer, 1993.

[21] Nicole Kobie. Who do you blame when an algorithm gets you fired? http://www.wired.co.uk/article/make-algorithms-accountable, January 2016. (Accessed on 06/18/2016).

[22] Claire Cain Miller. Can an algorithm hire better than a human? http://www.nytimes.com/2015/06/26/upshot/can-an-algorithm-hire-better-than-a-human.html, June 2015. (Accessed on 06/18/2016).

[23] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI*, pages 2476–2482, 2014.

[24] Executive Office of the President. Big data: Seizing opportunities, preserving values. https://www.whitehouse.gov/sites/default/files/

docs/big_data_privacy_report_may_1_2014.pdf, May 2014. (Accessed on 06/18/2016).

[25] Walt L Perry. *Predictive policing: The role of crime forecasting in law enforcement operations*. Rand Corporation, 2013.

[26] Latanya Sweeney. Discrimination in online ad delivery. *Queue*, 11(3):10, 2013.

[27] Andrew Tutt. An fda for algorithms. *Available at SSRN 2747994*, 2016.

[28] Jennifer Valentino-Devries, Jeremy Singer-Vine, and Ashkan Soltani. Websites vary prices, deals based on users' information. http://www.wsj.com/articles/SB10001424127887323777204578189391813881534, December 2012. (Accessed on 06/18/2016).

[29] Richard S. Zemel, Yu Wu, Kevin Swersky, Toniann Pitassi, and Cynthia Dwork. Learning fair representations. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 325–333, 2013.