Seldonian Toolkit: Building Software with Safe and Fair Machine Learning

Austin Hoag  
Berkeley Existential Risk Initiative  
USA  
austinthomashoag@gmail.com

James E. Kostas, Bruno Castro da Silva, Philip S. Thomas, Yury Brun  
University of Massachusetts  
USA  
{jekostas, bsilva, pthomas, brun}@cs.umass.edu

Abstract—We present the Seldonian Toolkit, which enables software engineers to integrate provably safe and fair machine learning algorithms into their systems. Software systems that use data and machine learning are routinely deployed in a wide range of settings from medical applications, autonomous vehicles, the criminal justice system, and hiring processes. These systems, however, can produce unsafe and unfair behavior, such as suggesting potentially fatal medical treatments, making racist or sexist predictions, or facilitating radicalization and polarization. To reduce these undesirable behaviors, software engineers need the ability to easily integrate their machine-learning-based systems with domain-specific safety and fairness requirements defined by domain experts, such as doctors and hiring managers. The Seldonian Toolkit provides special machine learning algorithms that enable software engineers to incorporate such expert-defined requirements of safety and fairness into their systems, while provably guaranteeing those requirements will be satisfied. A video demonstrating the Seldonian Toolkit is available at https://youtu.be/wHR-hDm9jX4.

I. INTRODUCTION

The use of machine learning (ML) algorithms has become increasingly commonplace, with a wide range of applications including optimizing user experiences [44], providing decision support for high-risk high-impact applications such as criminal sentencing [5], deciding which loans should be approved [6], deciding which resumes should be evaluated by a human [37], and providing medical decision support [29].

Unfortunately, data-driven software can sometimes produce undesirable behavior, such as unsafe or unfair behavior. IBM Watson, for example, recommended potentially fatal cancer treatments [36], and ML software used in 11 US states to predict whether a person will commit a crime in the future was found to have a racial bias [5].

One of the root causes of these undesirable behaviors is the fact that there is a disconnect between the users and the developers of data-driven software. Users, such as doctors, lawyers, and hiring managers, have the expertise to define what unsafe or unfair behavior means. However, these users are not the people building the software systems that must satisfy safety and fairness requirements. The software engineers that do build such systems, by contrast, are typically neither domain experts nor ML experts, so it is critical that they have the ability to integrate ML algorithms and domain-specific safety and fairness requirements.

We introduce the Seldonian¹ Toolkit, a framework that bridges the gap between ML experts, software engineers, and users. The toolkit implements a Seldonian ML algorithm [45] that allows domain-expert users to specify safety and fairness requirements, and trains ML models that are probabilistically verified to satisfy those requirements. The Seldonian Toolkit’s key contributions are the support for: (1) specification of custom, domain-specific constraints that can encode safety and fairness properties, (2) training ML models while providing high-confidence guarantees that these models, applied to new data, satisfy the specified safety or fairness constraints, and (3) evaluating the effectiveness of Seldonian algorithms for a given use case via comparison to standard ML approaches and other fairness-aware ML algorithms.

II. THE SELDONIAN TOOLKIT

The Seldonian Toolkit consists of two Python libraries and a graphical user interface that runs in the browser. The Python libraries are the Seldonian Engine² and the Seldonian Experiments Library³. The graphical user interface is called the Seldonian Interface GUI (SIGUI)⁴. We first describe the SIGUI (Section II-A), the Seldonian Engine (Section II-B), and then the Experiments library (Section II-C).

With beginners in mind, we provide tutorials and examples (with more in development) on how to use the Seldonian Toolkit, starting from installation and progressing toward real-world end-to-end use cases, such as creating safe and fair deep learning and computer vision models with the toolkit.⁵

We have developed the toolkit with ease of adoption in mind, interfacing with ubiquitous tools, such as NumPy, SciPy, scikit-learn, and PyTorch. Currently, six groups of computer science graduate students at the University of Massachusetts are applying the toolkit to distinct ML problems, providing us with feedback.

¹The toolkit name [45] is a homage to Isaac Asimov’s fictional character, Hari Seldon, a resident of a universe where Asimov’s three laws of robotics fail to adequately control agent behavior due to their non-probabilistic requirements, and who formulated and solved a machine learning problem that would likely have required probabilistic constraints [7].
²https://github.com/seldonian-toolkit/Engine
³https://github.com/seldonian-toolkit/Experiments
⁴https://github.com/seldonian-toolkit/GUI
⁵https://seldonian.cs.umass.edu/Tutorials/tutorials/
**Behavioral Constraints:**

<table>
<thead>
<tr>
<th>Constraint #1:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Constraint Diagram" /></td>
<td></td>
</tr>
</tbody>
</table>

The SIGUI is designed to help users specify safety and fairness behavioral constraint requirements. The form of these constraints can vary depending on the domain, and we designed the Seldonian Toolkit to be flexible and accept a wide range of such requirements. Many prominent definitions of fairness [40] can be defined as mathematical statements or inequalities. For example, a common desirable fairness definition, known as demographic parity, requires that a positive outcome be given to the same fraction of people of two protected classes. Assume, for instance, that an ML model is designed to decide who should get a loan while not discriminating based on race. In this case, the same fraction of applicants of each race should be given loans, up to some threshold. Not all types of safety constraints can be expressed as mathematical statements. Consider, for example, a user observing a robotic system performing an unwanted behavior, such as a chess robot breaking a child’s finger [9]. The mathematical expression required to instruct the robot not to do this is hard to define analytically by non-expert users. The Seldonian Toolkit allows, nonetheless, to allow users to specify that this behavior is undesirable.

Suppose the user wishes to specify that an application must satisfy the demographic parity requirement. The toolkit’s behavioral constraint specifications consist of two parts: (1) constraint strings and (2) confidence levels. Constraint strings are mathematical inequalities that must hold. For example, Figure 1 shows how the constraint string for demographic parity can be specified in the SIGUI. The user first selects “measure functions.” These include metrics quantifying properties of a given ML model, such as its positive rate (PR), negative rate (NR), false-positive rate (FPR), etc. Then, the user combines the measure functions with mathematical functions (e.g., max() and operators (e.g., ratio), and associates them with protected classes, such as gender. For example, in Figure 1, “PR | [Male]” means “positive rate for men.” To specify demographic parity with respect to (binary) gender, the user specifies “abs((PR | [Male]) − (PR | [Female])) − 0.15 ≤ 0,” which means “the absolute value of the difference between positive rates for men and women must be less than or equal to 0.15.” The toolkit’s probabilistic verification requires the user to also specify, for each constraint, the required confidence level, δ. This δ is the acceptable probability of violation of the constraint. The user could, for example, require the Seldonian Toolkit to produce an ML model with at least 99.9% confidence (δ = 0.001) that it will satisfy the above-specified demographic parity constraint when applied to unseen data.

Safety constraints can be defined similarly to fairness constraints. For example, a behavioral constraint on an insulin pump ML-model update could require that the update cause no more instances of hypoglycemia than the original version [45], or that a new model’s accuracy is strictly higher, with high confidence.

To simplify the constraint specification process, SIGUI allows the user to select from five commonly-used predefined fairness constraints, modify them, or build their own completely unique constraints using drag-and-drop building blocks. SIGUI also includes a tutorial to speed up the learning process. The user can specify an unlimited number of behavioral constraint requirements for each model.

**B. Seldonian Engine**

The Seldonian Engine is the core library of the toolkit and implements a general-purpose Seldonian algorithm. Given a dataset D and a set of behavioral constraint requirements (recall Section II-A), a developer can use the Engine to train an ML model that is probabilistically guaranteed to satisfy all of the requirements. (Note that under certain conditions, such as when given insufficient data to train a model, or contradictory requirements, the Engine will explicitly state that no solution could be found, as further discussed below.)

While the Seldonian algorithm [45] is not a contribution of this work (the Toolkit is a usable implementation of a previously-published algorithm), we briefly describe it here. At a high level, the algorithm operates as follows. First, the available dataset D is partitioned into two disjoint sets: candidate data, Dcand, and safety data, Dsafety. Dcand is provided as input to a component called candidate selection, which trains a single ML model that the algorithm plans to return. This model is called the candidate solution. The algorithm attempts to select a model that maximizes performance, while also satisfying the behavioral constraints, but, at this point, provides no guarantees. Next, the safety test component probabilistically verifies that the candidate solution satisfies the behavioral constraints. The safety test executes the

---

Fig. 2. The Seldonian Experiments Library can help software engineers evaluate trade-offs between using Seldonian and non-Seldonian ML algorithms. Here, models trained using the Seldonian Toolkit are less accurate than ones trained using standard linear regression, and require more data, but always satisfy a fairness requirement that the linear regression model fails.

solution on $D_{\text{safety}}$ and checks if the constraints hold on that data. The size of $D_{\text{safety}}$ is selected specifically to allow the use of probability bounds, such as Hoeffding's inequality, to estimate the likelihood that the requirement is satisfied not only on $D_{\text{safety}}$, but also on new, unseen data. If the bound satisfies the specified confidence level, the algorithm returns the candidate model. Otherwise, if the model fails the requirement or the algorithm's confidence is insufficient, the algorithm returns “No Solution Found.” Our prior work has formally proven that the produced models are probabilistically verified to satisfy the provided safety and fairness constraints and empirically compared Seldonian algorithms to other fairness-aware ML algorithms [19], [31], [45].

To use the Seldonian Engine, the developer needs a set of behavioral constraint requirements (created by, or in coordination with a domain expert), a dataset, and instructions on which ML model the Engine should use internally. The Engine can use any (parametric) ML model, such as a deep neural network or a linear regression model. The Engine uses its implementation of the Seldonian algorithm to train and probabilistically verify the model, and either returns that model or “No Solution Found.”

C. Seldonian Experiments Library

Developers and domain-expert users will understandably want to know how well their Seldonian algorithms perform compared to existing ML models that do not necessarily enforce safety or fairness. Satisfying behavioral constraint requirements (created by, or in coordination with a domain expert), a dataset, and instructions on which ML model the Engine should use internally. The Engine can use any (parametric) ML model, such as a deep neural network or a linear regression model. The Engine uses its implementation of the Seldonian algorithm to train and probabilistically verify the model, and either returns that model or “No Solution Found.”

Developers and domain-expert users will understandably want to know how well their Seldonian algorithms perform compared to existing ML models that do not necessarily enforce safety or fairness. Satisfying behavioral constraint requirements (created by, or in coordination with a domain expert), a dataset, and instructions on which ML model the Engine should use internally. The Engine can use any (parametric) ML model, such as a deep neural network or a linear regression model. The Engine uses its implementation of the Seldonian algorithm to train and probabilistically verify the model, and either returns that model or “No Solution Found.”

To use the Seldonian Engine, the developer needs a set of behavioral constraint requirements (created by, or in coordination with a domain expert), a dataset, and instructions on which ML model the Engine should use internally. The Engine can use any (parametric) ML model, such as a deep neural network or a linear regression model. The Engine uses its implementation of the Seldonian algorithm to train and probabilistically verify the model, and either returns that model or “No Solution Found.”

The Experiments Library helps the developer understand these trade-offs. Consider an engineer tasked with building a system that predicts housing prices within two zip codes. The developer has access to historical data describing how various features, including zip code, size, and the year when it was built, influence housing prices. The stakeholders (or, perhaps, the law) require that the system must have similar accuracy in both zip codes, perhaps because these zip codes correlate with the residents’ races, and failing this requirement could lead to discrimination. The developer could use a standard ML model, e.g., linear regression implemented in scikit-learn [35], to achieve good overall accuracy in predicting house prices, but they might find that their predictions are more accurate in one zip code than the other.

The developer can input that scikit-learn model and dataset into the Seldonian Toolkit, along with the behavioral constraint from the stakeholders (formalized using SIGUI). The Engine will train a new model that satisfies the requirement, i.e., it will be similarly accurate in both zip codes. Then, the Experiments Library will produce a set of three diagnostic plots to help the developer understand the potential trade-offs.

We generated a synthetic dataset with a single feature (house size), a single sensitive attribute (zip code), and a single label (house price), and then used the Experiments Library on this dataset with a fairness constraint that enforces similar accuracy in the two zip codes. Figure 2 shows the resulting three plots for this house-price prediction problem. (While the Experiments Library currently makes these plots static, future work can explore if interactive features can help users better understand model behavior.) The left plot presents the accuracy of the two ML models (produced by scikit-learn and the Seldonian algorithm) in terms of mean squared error, as a function of the size of the dataset. The larger the mean squared error, the less accurate the model is in explaining the data. Here, the Seldonian model sacrifices accuracy to be able to satisfy the
fairness constraint with high confidence. Note that the plot shows no accuracy for datasets smaller than $10^3$ data points, as we explain next. Here, we stress, again, that while we selected this example to showcase the possible trade-offs that may occur, prior research has shown that accuracy can often remain just as high as that of standard, unsafe ML algorithms when safety or fairness constraints are enforced [17, 45].

The middle plot shows how often the algorithm produces a verified fair model (as opposed to “No Solution Found”), as a function of the size of the dataset. While scikit-learn always produces a model (a very inaccurate one for small datasets), the Seldonian Engine is only able to produce models it can confidently say are fair once it has access to approximately $10^3$ data points. (This is the reason the left plot did not contain accuracy information for small datasets). The amount of needed data is another important trade-off factor in choosing algorithms.

Finally, the right plot shows how often the trained models violate the behavioral constraint requirement when applied to new, unseen data. Scikit-learn’s linear regression model always violates the fairness constraint, thus discriminating against one zip code. The Seldonian algorithm’s model, by contrast, never violates the constraint.

Importantly, external ML models can be used in the toolkit, with little additional configuration. Because reimplementing ML models in a new framework, especially complex ones such as deep neural networks, can be prohibitively time consuming when designing software, the Seldonian Toolkit supports NumPy [20], SciPy [47], scikit-learn [35], and PyTorch models [34], and work is ongoing to add support for Keras, TensorFlow, and other popular ML libraries. Further, the Experiments Library incorporates Microsoft Fairlearn’s [2] fairness-aware classification algorithms, and we similarly plan to add support for the Fairness Constraints [51] library. This allows developers to directly compare their Seldonian models to other methods that aim to improve fairness (but which do not provide similar guarantees).

III. RELATED WORK

Recent research has argued that fairness is not only an important aspect of machine learning, but must also be addressed within the software engineering community, developing tools and technology to support software engineers in building fair systems [11]. Several frameworks exist to help data scientists evaluate models with respect to their fairness and other performance measures, including Fairlearn [10], Fairkit-learn [26], [27], AIFairness 360 [24], FairViz [12], FairML [1], and Fairway [13]. Like our Seldonian Toolkit, these frameworks help evaluate trade-offs in models but none provide methods for training provably fair or safe ML models. Meanwhile, algorithms aimed to enforce fairness in ML models, e.g., [2], [51], or repair models to reduce bias, e.g., [22], [42], similarly do not provide the kinds of high-confidence guarantees the Seldonian Toolkit provides. In fact, two recent systematic literature reviews of research on testing models and systems for fairness [14] and mitigating bias [21], found that other than Seldonian algorithms [45], none provide high-confidence guarantees.

Formal verification of software systems using proof assistants, such as Coq [43] and Isabelle/HOL [32], allows proving software properties but, unlike our work, requires significant manual effort [28], [30]. Recent research has enabled fully automating formal verification by learning models of existing proofs for a set of systems and then using those models to generate proofs of new properties for new systems [3], [8], [15], [16], [18], [23], [25], [33], [38], [39], [50]. However, these approaches do not apply directly to machine learning models and, typically, require the system whose properties are being verified to be written in the language used by the underlying proof assistant.

Frameworks such as FairPrep [41] and FairRover [52] help data scientists follow best practices in software engineering and ML, focusing on responsible and ethical ML uses. Meanwhile, Google’s What-If tool helps scientists analyze and understand ML models without writing code [49]. These tools are complementary to our work.

An important complement to building systems that enforce fairness is testing systems for discrimination [4], [17], [46]. When additional data become available, these tools can be used to audit models learned by the Seldonian Toolkit and by other methods.

Finally, Seldonian algorithms have been developed for contextual bandits [31], the setting where the training data and deployment data come from different distributions [19], and to enforce measures of long-term fairness [48], suggesting future extensions of the Seldonian Toolkit.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grants no. CCF-1763423, CCF-2018372, and CCF-2210243, and by a gift from the Berkeley Existential Risk Initiative.

REFERENCES


For more information, please visit the project website: https://coq.inria.fr