

O'REILLY®

Machine Learning for High-Risk Applications

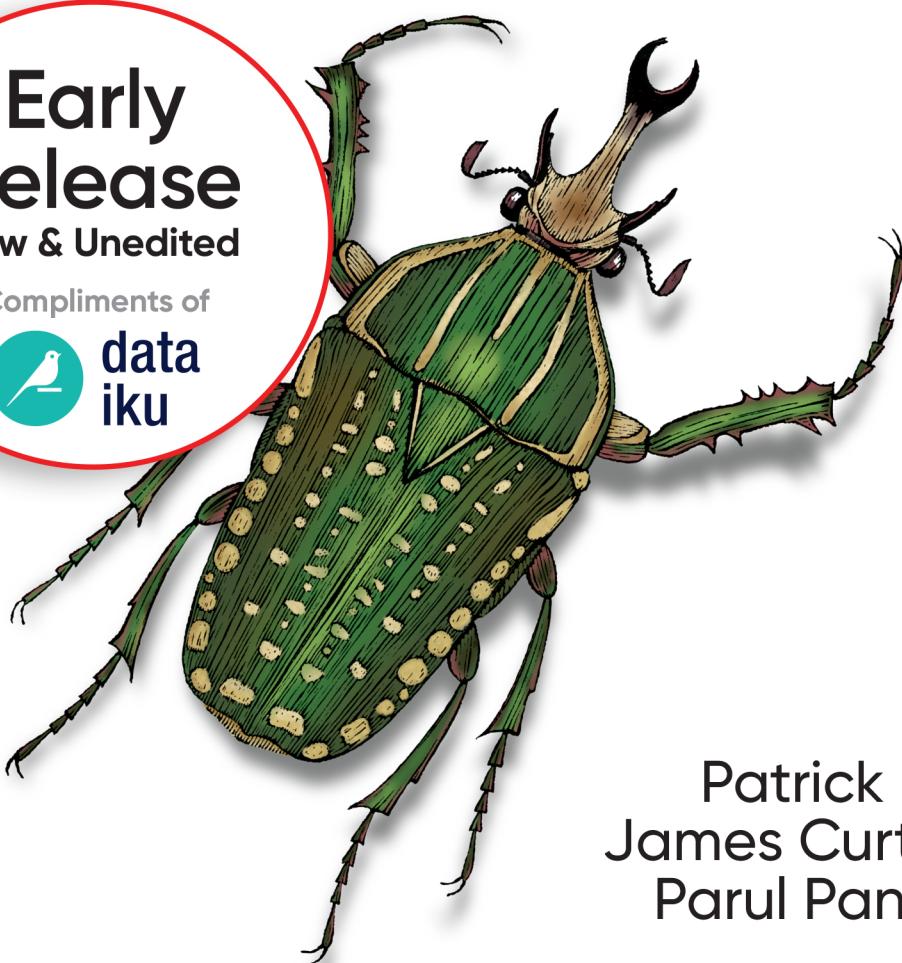
Techniques for Responsible AI

Early
Release
Raw & Unedited

Compliments of



data
iku



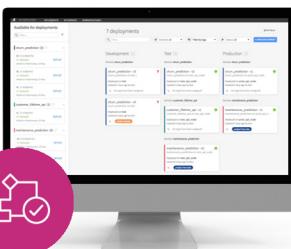
Patrick Hall,
James Curtis &
Parul Pandey

AI Governance With Dataiku



The Platform for Everyday AI

Leveraging one central solution for AI means more transparent, repeatable, and scalable AI. Dataiku gives people (whether technical and working in code or on the business side and low- or no-code) the ability to make better day-to-day decisions with data.



Deploy, Monitor, & Manage Machine Learning Projects in Production

Keep critical AI initiatives up and running with Dataiku, which offers model monitoring, drift detection, automatic model retraining, easy production project model updates, automated CI/CD, and more.



Manage Risk & Ensure Compliance at Scale

Advanced AI Governance including customizable governance plans, production sign-off, risk and value analysis together with permissions management, user directory integration, audit trails, and secure API access make Dataiku the perfect choice for streamlined AI Governance across the organization.



Understand Outputs, Increase Trust, & Identify Potential Bias

Dataiku provides critical capabilities for explainable and responsible AI, including reports on feature importance, partial dependence plots, subpopulation analysis, model fairness, and individual prediction explanations.

Machine Learning for High-Risk Applications

Techniques for Responsible AI

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Patrick Hall, James Curtis, and Parul Pandey

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Machine Learning for High-Risk Applications

by Patrick Hall

Copyright © 2022 Patrick Hall, James Curtis, and Parul Pandey. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Michele Cronin and Rebecca Novack

Indexer: FILL IN INDEXER

Production Editor: Beth Kelly

Interior Designer: David Futato

Copyeditor: FILL IN COPYEDITOR

Cover Designer: Karen Montgomery

Proofreader: FILL IN PROOFREADER

Illustrator: Kate Dullea

June 2023: First Edition

Revision History for the Early Release

2021-05-26: First Release

2021-08-19: Second Release

2021-10-04: Third Release

2021-12-08: Fourth Release

2022-02-08: Fifth Release

2022-06-08: Sixth Release

2022-08-08: Seventh Release

2022-09-28: Eighth Release

2022-12-02: Ninth Release

2023-01-11: Tenth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098102432> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Machine Learning for High-Risk Applications*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Dataiku. See our [statement of editorial independence](#).

978-1-098-10243-2

[FILL IN]

Table of Contents

Preface.....	xı
1. Contemporary Machine Learning Risk Management.....	21
A Snapshot of the Legal and Regulatory Landscape	22
The Proposed EU AI Act	23
US Federal Laws and Regulations	23
State and Municipal Laws	24
Basic Product Liability	24
Federal Trade Commission Enforcement	26
Authoritative Best Practices	27
AI Incidents	30
Cultural Competencies for Machine Learning Risk Management	32
Organizational Accountability	32
Culture of Effective Challenge	33
Diverse and Experienced Teams	33
Drinking Our Own Champagne	34
Moving Fast and Breaking Things	34
Organizational Processes for Machine Learning Risk Management	35
Forecasting Failure Modes	35
Model Risk Management Processes	36
Beyond Model Risk Management	41
Case Study: The Rise and Fall of Zillow's iBuying	45
Fallout	46
Lessons Learned	46
Resources	49
2. Interpretable and Explainable Machine Learning.....	51
Important Ideas for Interpretability and Explainability	52

Explainable Models	57
Additive Models	57
Decision Trees	62
An Ecosystem of Explainable Machine Learning Models	64
Post-hoc Explanation	68
Feature Importance	68
Surrogate Models	80
Linear Models and Local Interpretable Model-agnostic Explanations	82
Plots of Model Performance	84
Cluster profiling	87
Stubborn Difficulties of Post-hoc Explanation in Practice	87
Pairing Explainable Models and Post-hoc Explanation	91
Case Study: Graded by Algorithm	93
Resources	96
3. Debugging Machine Learning Systems for Safety and Performance.....	97
Training	99
Reproducibility	99
Data Quality	101
Model Specification for Real-world Outcomes	104
The Future of Safe and Robust Machine Learning	107
Model Debugging	108
Software Testing	108
Traditional Model Assessment	109
Machine Learning Bugs	111
Residual Analysis	119
Sensitivity Analysis	122
Benchmark Models	124
Remediation: Fixing Bugs	125
Deployment	127
Domain Safety	128
Model Monitoring	129
Case Study: Death by Autonomous Vehicle	133
Fallout	133
An Unprepared Legal System	133
Lessons Learned	134
Resources	135
4. Managing Bias in Machine Learning.....	137
ISO and NIST Definitions for Bias	139
Systemic Bias	139
Statistical Bias	140

Human Biases and Data Science Culture	140
United States Legal Notions of ML Bias	142
Who Tends to Experience Bias from ML Systems	144
Harms That People Experience	146
Testing for Bias	147
Testing Data	148
Traditional Approaches: Testing for Equivalent Outcomes	150
A New Mindest: Testing for Equivalent Performance Quality	153
On the Horizon: Tests for the Broader ML Ecosystem	155
Summary Test Plan	158
Mitigating Bias	159
Technical Factors in Mitigating Bias	160
The Scientific Method and Experimental Design	160
Bias Mitigation Approaches	161
Human Factors in Mitigating Bias	165
Case Study: The Bias Bug Bounty	168
Resources	170
5. Security for Machine Learning.....	171
Security Basics	172
The Adversarial Mindset	172
CIA Triad	173
Best Practices for Data Scientists	175
Machine Learning Attacks	177
Integrity Attacks: Manipulated Machine Learning Outputs	177
Confidentiality Attacks: Extracted Information	182
General AI Security Concerns	185
Counter-measures	187
Model Debugging for Security	187
Model Monitoring For Security	190
Privacy-enhancing Technologies	191
Robust Machine Learning	193
General Countermeasures	194
Case Study: Real-world Evasion Attacks	133
Lessons Learned	196
Resources	197
6. Explainable Boosting Machines and Explaining XGBoost.....	199
Concept Refresher: ML Transparency	200
Additivity vs. Interactions	200
Steps Toward Causality with Constraints	201
Partial Dependence and Individual Conditional Explanation	201

Shapley Values	203
Model Documentation	204
The GAM Family of Interpretable Models	205
Elastic Net Penalized GLM w/ Alpha and Lambda Search	205
Generalized Additive Models	209
GA2M and Explainable Boosting Machines	212
XGBoost with Constraints and Explainable Artificial Intelligence	214
Constrained and Unconstrained XGBoost	215
Explaining Model Behavior with Partial Dependence and ICE	219
Decision Tree Surrogate Models as an Explanation Technique	221
Shapley Value Explanations	225
Resources	233
7. Debugging a PyTorch Image Classifier.....	235
Concept Refresher: Debugging Deep Learning	236
Debugging a PyTorch Image Classifier	238
Data Quality and Leaks	239
Software Testing for Deep Learning	241
Sensitivity Analysis for Deep Learning	242
Remediation	251
Conclusion	260
Resources	261
8. Testing and Remediating Bias with XGBoost.....	263
Concept Refresher: Managing ML Bias	264
Model Training	267
Evaluating Models for Bias	272
Testing Approaches for Groups	272
Individual Fairness	281
Proxy Bias	285
Remediating Bias	286
Pre-processing	286
In-processing	290
Post-processing	295
Model Selection	297
Conclusion	301
Resources	303
9. Red-teaming XGBoost.....	305
Concept Refresher	306
CIA Triad	306
Attacks	307

Countermeasures	309
Model Training	310
Attacks for Red-teaming	314
Model Extraction Attacks	314
Adversarial Example Attacks	318
Membership Attacks	320
Data Poisoning	321
Backdoors	324
Conclusion	328
Resources	329

Preface

Today, machine learning (ML) is the most commercially viable sub-discipline of artificial intelligence (AI). ML systems are used to make high-risk decisions in employment, bail, parole, lending, security and in many other high-impact applications throughout the world's economies and governments. In a corporate setting, ML systems are used in all parts of an organization — from consumer-facing products, to employee assessments, to back-office automation, and more. Indeed, the past decade has brought with it even wider adoption of ML technologies. But it has also proven that ML presents risks to its operators, consumers, and even the general public.

Like all other technologies, ML can fail — whether by unintentional misuse or intentional abuse. As of today, there have been thousands of public reports of algorithmic discrimination, data privacy violations, training data security breaches and other harmful incidents. Such risks must be mitigated before organizations, and the public, can realize the true benefits of this exciting technology. Addressing ML's risks requires action from practitioners. While nascent standards, to which this book aims to adhere, are beginning to take shape, the practice of ML is still lacks broadly accepted professional licensing or best practices. That means it's largely up to individual practitioners to hold themselves accountable for the good and bad outcomes their technology may evoke when deployed into the world. *Machine Learning for High-Risk Applications* will arm practitioners with a solid understanding of model risk management processes and new ways to use common Python tools for training explainable models and debugging them for reliability, safety, bias management, security and privacy issues.



We adapt a definition of AI from Stuart Russel and Peter Norvig's, *Artificial Intelligence: A Modern Approach*: The designing and building of intelligent systems that receive signals from the environment and take actions that affect that environment (2020). For ML, we use the common definition attributed — perhaps apocryphally — to Arthur Samuel: [A] field of study that gives computers the ability to learn without being explicitly programmed (*circa* 1960).

Who Should Read This Book

This is a mostly technical book for early-to-middle career ML engineers and data scientists who want to learn about responsible use of ML or ML risk management. The code examples are written in Python. That said, this book probably isn't for every data scientist and engineer out there coding in Python. This book is for you if you want to learn some model governance basics and update your workflow to accommodate basic risk controls. This book is for you if your work needs to comply with certain nondiscrimination, transparency, privacy or security standards. (Although we can't guarantee compliance or provide legal advice!). This book is for you if you want to train explainable models, and learn to edit and debug them. Finally, this book is for you if you're concerned that your work in ML may be leading to unintended consequences relating to sociological bias, data privacy violations, security vulnerabilities, or other known problems caused by automated decision-making writ large, and you want to do something about it.

Of course, this book may be of interest to others. If you're coming to ML from a field like physics, econometrics or psychometrics, this book can help you learn how to blend newer ML techniques with established domain expertise and notions of validity or causality. This book may give regulators or policy professionals some insights into the current state of ML technologies that would be used in an effort to comply with laws, regulations, or standards. Technical risk executives or risk managers may find this book helpful in providing an updated overview of newer ML approaches suited for high-stakes applications. And expert data scientists or ML engineers may find this book educational too, but they may also find it challenges many established data science practices.

What Readers Will Learn

Readers of this book will be exposed to both traditional model risk management and how to blend it with computer security best practices, like incident response, bug bounties, and red-teaming, to apply battle-tested risk controls to ML workflows and systems. This book will introduce a number of older and newer explainable models, and explanation techniques that make ML systems even more transparent. Once

we've setup a solid foundation of highly transparent models, we'll dig into testing models for safety and reliability. That's a lot easier when we can see how our model works! We'll go way beyond quality measurements in holdout data to explore how to apply well-known diagnostic techniques like residual analysis, sensitivity analysis, and benchmarking to new types of ML models. We'll then progress to structuring models for bias management, testing for bias, and remediating bias from an organizational and technical perspective. Finally, we'll discuss security for ML pipelines and APIs.

Alignment with the NIST AI Risk Management Framework

In an attempt to follow our own advice, and make the book even more practical for those working on high-risk applications, we will highlight where the proposed approaches in the book align to the nascent National Institute of Standards and Technology (NIST) AI Risk Management Framework (RMF). Application of external standards is a well-known risk management tactic, and NIST has an incredible track record for authoritative technical guidance. The AI RMF has many components but two of the most central are the characteristics for trustworthiness in AI and the core RMF guidance. The characteristics for trustworthiness establish the basic principles of AI risk management, while the core RMF guidance provides advice for implementation of risk controls. We will use vocabulary relating to NIST's characteristics for AI trustworthiness throughout the book: validity, reliability, safety, security, resiliency, transparency, accountability, explainability, interpretability, bias management and enhanced privacy. At the beginning of each chapter in Part One we'll also use a call out box to breakdown how and where the content aligns to specific aspects of the core NIST AI RMF map, measure, manage, and govern functions. We hope alignment to the NIST AI RMF improves the usability of the book, making the book a more effective AI risk management tool.

Preliminary Book Outline

The book is broken into two parts. Part One discusses issues from a practical application perspective, with dashes of theory where necessary. Part Two contains long-form Python coding examples, addressing the topics in Part One from both structured and unstructured data perspectives.

Part One

Part One begins with a deep dive into pending regulations, discussions of product liability, and a thorough treatment of traditional model risk management. Because many of these practices assume a somewhat staid and professional approach to modeling — a far cry from today's common “move fast and break things” ethos — we'll also discuss how to incorporate computer security best practices that assume failure

into model governance. Chapter Two presents the burgeoning ecosystem of explainable models. We cover the generalized additive model (GAM) family in the most depth, but also discuss many other types of high-quality and high-transparency estimators. Chapter Two also outlines many different post-hoc explanation techniques, but with an eye toward rigor and known problems with this somewhat over-hyped sub-field of responsible ML techniques. Chapter Three tackles model validation, but in a way that actually tests model's assumptions and real-world reliability. We'll go over software testing basics as well as touching on highlights from the new field of model debugging. Chapter Four overviews the *socio-technical* aspects of fairness and bias in ML before transitioning to technical bias measurement and remediation approaches. Chapter Four then treats bias testing in some detail, including tests for disparate impact, differential validity, and uncovering drivers of bias with Shapley values and other explanation techniques. Chapter Four also addresses both established and conservative methods for bias remediation, and more cutting edge dual-objective, adversarial, and pre-, in-, and post-processing remediation techniques. Chapter Five closes Part One by laying out how to red-team ML systems, starting with the basics of computer security and moving into discussions of common ML attacks, adversarial ML, and robust ML. Each chapter in Part One closes with a cases discussion relating to topics like Zillow's ibuying meltdown, the A-levels scandal in the UK, the fatal crash of a self-driving Uber, Twitter's inaugural bias bug bounty, and real-world ML evasion attacks. Each chapter will also outline alignment between content and the NIST AI RMF.

Part Two

Part Two expands on the ideas in Part One with a series of thorough code example Chapters. Chapter Six puts explainable boosting machines (EBM), XGBoost, and explainable AI techniques through their paces in a basic consumer finance example. Chapter Seven applies post-hoc explanation techniques to a PyTorch image classifier. In Chapter Eight, we'll debug our consumer finance models for performance problems, and do the same for our image classifier in Chapter Nine. Chapter Ten contains detailed examples relating to bias testing and bias remediation, and Chapter Eleven provides examples of ML attacks and countermeasures for tree-based models.

How to Succeed in High-risk Applications

We end the book in Chapter Twelve with more general advice for how to succeed in high-risk ML applications. It's not by moving fast and breaking things. For some low risk use cases, it might be fine to apply a quick and dirty approach. But as ML becomes more regulated and is used in more high-risk applications, the consequences of breaking things are becoming more serious. Chapter 12 caps off the book with hard-won practical advice for applying ML in high-stakes scenarios.

Our hope with the first edition of this text is to provide a legitimate alternative to the standard black-box, compressed-time-frame workflows that are common in ML today. This book should provide a set of vocabulary, ideas, tools, and techniques that enable practitioners to be more deliberate in their very important work.

Example Datasets

We rely on two primary datasets in this book, to explain techniques or to demonstrate approaches and discuss their results. For the structured data chapters, we used a slightly modified version of the Taiwan Credit Data available from University of California Irvine [Machine Learning Repository](#) or [Kaggle](#). These are example datasets, not fit for training models in high-risk applications, but they are well known and easily accessible. Their shortcomings also allow us to point out various data, modeling, and interpretation pitfalls. We will refer to these datasets many times in subsequent chapters, so be sure to get a feel for them before diving into the rest of the book.

Taiwan Credit Data

The credit card default data contains demographic and payment information about credit card customers in Taiwan in the year 2005. In general, the goal with this dataset is to use past payment statuses (PAY_*), past payment amounts (PAY_AMT*), and bill amounts (BILL_AMT*) as inputs to predict whether a customer will meet their next payment (DELINQ_NEXT = 0). We've added simulated SEX and RACE markers to this dataset to illustrate bias testing and remediation approaches. We use the payment information as input features, and following best practices to manage bias in ML systems, we do not use the demographic information as model inputs. The complete data dictionary is available below.

Name	Modeling Role	Measurement Level	Description
ID	ID	int	unique row identifier
LIMIT_BAL	input	float	amount of previously awarded credit
SEX	demographic information	int	1 = male; 2 = female
RACE	demographic information	int	1 = Hispanic; 2 = Black; 3 = White; 4 = Asian
EDUCATION	demographic information	int	1 = graduate school; 2 = university; 3 = high school; 4 = others
MARRIAGE	demographic information	int	1 = married; 2 = single; 3 = others
AGE	demographic information	int	age in years

Name	Modeling Role	Measurement Level	Description
PAY_0, PAY_2 - PAY_6	input	int	history of past payment; PAY_0 = the repayment status in September, 2005; PAY_2 = the repayment status in August, 2005; ...; PAY_6 = the repayment status in April, 2005. The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; ...; 8 = payment delay for eight months; 9 = payment delay for nine months and above
BILL_AMT1 - BILL_AMT6	input	float	amount of bill statement; BILL_AMT1 = amount of bill statement in September, 2005; BILL_AMT2 = amount of bill statement in August, 2005; ...; BILL_AMT6 = amount of bill statement in April, 2005
PAY_AMT1 - PAY_AMT6	input	float	amount of previous payment; PAY_AMT1 = amount paid in September, 2005; PAY_AMT2 = amount paid in August, 2005; ...; PAY_AMT6 = amount paid in April, 2005
DELINQ_NEXT	target	int	whether a customer's next payment is delinquent (late), 1 = late; 0 = on-time

As readers will see in the following chapters this dataset encodes some pathological flaws. It's too small to train usable high-capacity ML estimators, nearly all of the signal for `DELINQ_NEXT` is encoded in `PAY_0`, and even these kinds of straightforward consumer financial attributes encode surprising levels of systemic bias. As the book progresses, we'll attempt to deal with these issues and uncover others.

Kaggle Chest X-ray Data

For the deep learning chapters — Chapters [Chapter 6](#) and [Chapter 7](#) — we will be using the Kaggle [Chest X-ray images](#) dataset. This dataset is composed of roughly 5,800 images of two classes, `pneumonia` and `normal`. These labels were determined by human domain experts. The images are deidentified chest X-rays, taken during routine care visits to Guangzhou Women and Children's Medical Center in Guangzhou, China. See below for an example `pneumonia` image.

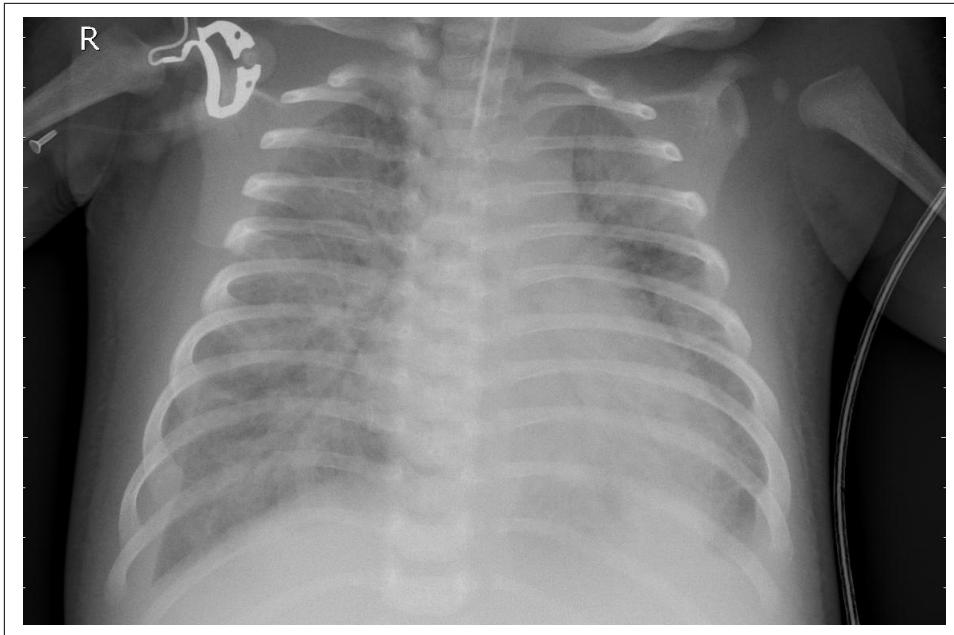


Figure P-1. An example pneumonia image from the Kaggle Chest X-ray dataset.

The main issues we'll face with this dataset are small size — even for transfer learning tasks, misalignment between the images in the dataset, visual artifacts that can give rise to shortcut learning, and the need for domain expertise to validate modeling outcomes. As with the Taiwan Credit data, we'll work through these issues and discover more in the later chapters of this book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic: Indicates new terms or important ideas.

Constant width: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a general note or suggestion.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Book Title by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Contemporary Machine Learning Risk Management

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Building the best machine learning (ML) system starts with cultural competencies and business processes. Chapter 1 presents numerous cultural and procedural approaches we can use to improve ML performance and safeguard our organization’s ML against real-world safety and performance problems. It also includes a case that illustrates what happens when an ML system is used without proper human oversight. The primary goal of the approaches discussed in this chapter is to create better ML systems. This might mean improved *in silica* test data performance. But it really means building models that perform as expected once deployed *in vivo*, so we don’t lose money, hurt people or cause other harms.



In vivo is Latin for “within the living.” We’ll sometimes use this term to mean something closer to “interacting with the living,” as in how ML models perform in the real-world when interacting with human users. *In silica* means “by means of computer modeling or computer simulation,” and we’ll use this term to describe the testing data scientists often perform in their development environments before deploying ML models.

The chapter begins with a discussion of the current legal and regulatory landscape for ML and some nascent best practice guidance, to inform system developers of their fundamental obligations when it comes to safety and performance. We’ll also introduce how the book aligns to the National Institute for Standards and Technology (NIST) [AI Risk Management Framework](#) (RMF) in this part of the chapter. Because those who do not study history are bound to repeat it, [Chapter 1](#) then highlights AI incidents, and discusses why understanding AI incidents is important for proper safety and performance in ML systems. Since many ML safety concerns require thinking beyond technical specifications, the chapter then blends model risk management (MRM), information technology (IT) security guidance, and practices from other fields to put forward numerous ideas for improving ML safety culture and processes within organizations. [Chapter 1](#) will close with a case study focusing on safety culture, legal ramifications, and AI incidents.

None of the risk management approaches discussed in this chapter is a silver bullet. If we want to manage risk successfully, we’ll need to pick from the wide variety of available controls those that work best for our organization. Larger organizations will typically be able to do more risk management than smaller organizations. Readers at large organizations may be able to implement many controls across various departments, divisions, or internal functions. Readers at smaller organizations will have to choose their risk management tactics judiciously. In the end, a great deal of technology risk management comes down to human behavior. Whichever risk controls an organization implements, they’ll need to be paired with strong governance and policies for the **people** that build and maintain ML systems.

A Snapshot of the Legal and Regulatory Landscape

It’s a myth that ML is unregulated. ML systems can and do break the law. Forgetting or ignoring legal context is one of the riskiest things an organization can do with respect to ML systems. That said, the legal and regulatory landscape for ML is complicated and changing quickly. This section aims to provide a snapshot of important laws and regulations for overview and awareness purposes. We’ll start by highlighting the pending EU AI Act. We’ll then discuss the many U.S. federal laws and regulations that touch on ML, US state and municipal laws for data privacy and AI, the basics of

product liability, and end the section with a rundown of recent Federal Trade Commission (FTC) enforcement actions.



The authors are not lawyers and nothing in this book is legal advice. The intersection of law and AI is an incredibly complex topic that data scientists and ML engineers are not equipped to handle alone. You may have legal concerns about ML systems that you work on. If so, seek real legal advice.

The Proposed EU AI Act

The EU has proposed sweeping regulations for AI that are expected to be passed in 2023. Known as the [EU AI Act](#) (AIA), they would prohibit certain uses of AI like distorting human behavior, social credit scoring, and real-time biometric surveillance. The AIA deems other uses as high risk, including applications in criminal justice, biometric identification, employment screening, critical infrastructure management, law enforcement, essential services, immigration, and others — placing a high documentation, governance, and risk management burden on these. Other applications would be considered limited or low risk, with fewer compliance obligations for their makers and operators. Much like the EU General Data Protection Regulation (GDPR) has changed the way companies handle data in the US and around the world, EU AI regulations are designed to have an out-sized impact on US and other international AI deployments. Whether we're working in the EU or not, we may need to start familiarizing ourselves with the AIA. One the best ways is to read the [Annexes](#), especially Annexes 1 and 3 — 8 that define terms and layout documentation and conformity requirements.

US Federal Laws and Regulations

As we've been using algorithms in one form or another for decades in our government and economy, many US federal laws and regulations touch on AI and ML. These regulations tend to focus on social discrimination by algorithms, but also treat transparency, privacy and other topics. The Civil Rights Acts of 1964 and 1991, the Americans with Disabilities Act (ADA), the Equal Credit Opportunity Act (ECOA), the Fair Credit Reporting Act (FCRA), and the Fair Housing Act (FHA) are some of the federal laws that attempt to prevent discrimination by algorithms in areas like employment, credit lending, and housing. ECOA, the Fair Credit Reporting Act (FCRA), and their more detailed implementation in Regulation B attempt to increase transparency in ML-based credit lending and guarantee recourse rights for credit consumers. For a rejected credit application, lenders are expected to indicate the reasons for the rejection, i.e., an *adverse action*, and describe the features in an ML model that drove the decision. If the provided reasoning or data is wrong, consumers should be able to appeal the decision.

The practice of MRM, defined in part in the Federal Reserve's SR 11-7 guidance, forms a part of regulatory examinations for large US banks, and sets up organizational, cultural, and technical processes for good and reliable performance of ML used in mission-critical financial applications. Much of this chapter is inspired by MRM guidance, as it's the most battle-tested ML risk management framework out there. Laws like the Health Insurance Portability and Accountability Act of 1996 (HIPAA) and Family Educational Rights and Privacy Act (FERPA) setup serious data privacy expectations in healthcare and for students. Like the GDPR, HIPAA's and FERPA's interactions with ML are material, complex, and still debated. These are not even all the US laws that might affect our use of ML, but hopefully this brief listing provides an idea of what the US federal government has decided is important enough to regulate.

State and Municipal Laws

US states and cities have also taken up laws and regulations for AI and ML. New York City (NYC) Local Law 144, which mandates bias audits for automated employment decision tools, is expected to go into effect in January of 2023. Under this law, every major employer in NYC will have to conduct bias testing of automated employment software and post the results on their website. Washington DC's proposed Stop Discrimination by Algorithms Act attempts to replicate federal expectations for non-discrimination and transparency, and for a much broader set of applications, for companies that operate in DC or use the data of many DC citizens.

Numerous states passed their own data privacy laws as well. Unlike the older HIPAA and FERPA federal laws, these state data privacy laws are often intentionally designed to partially regulate the use of AI and ML. States like California, Colorado, Virginia and others have passed data privacy laws that mention increased transparency, decreased bias, or both, for automated decision-making systems. Some states have put biometric or social media in their regulatory cross-hairs too. For example, Illinois' Biometric Information Privacy Act (BIPA) outlaws many uses of biometric data, and IL regulators have already started enforcement actions. The lack of a federal data privacy or AI law combined with this new crop of state and local laws makes AI and ML compliance landscape very complicated. Our uses of ML may or may not be regulated, or may be regulated to varying degrees, based on the specific application, industry and geography of the system.

Basic Product Liability

As makers of consumer products, data scientists and ML engineers have a fundamental obligation to create safe systems. To quote a recent Brookings Institute report, *Products liability law as a way to address AI harms*, “Manufacturers have an obligation to make products that will be safe when used in reasonably foreseeable ways. If an AI system is used in a foreseeable way and yet becomes a source of harm, a plaintiff

could assert that the manufacturer was negligent in not recognizing the possibility of that outcome.” Just like car or power tool manufacturers, makers of ML systems are subject broad legal standards for negligence and safety. Product safety has been the subject of large amounts of legal and economic analysis, but this subsection will focus on one of the first and simplest standards for negligence: the Hand Rule. Named after Judge Learned Hand, and coined in 1947, provides a viable framework for ML product makers to think about negligence and due diligence. The Hand Rule says that a product maker takes on a burden of care, and that the resources expended on that care should always be greater than the cost of a likely incident involving the product. Stated algebraically:

$$B \geq PL$$

In more plain terms, organizations are expected to apply care, i.e. time, resources, or money, to a level commensurate to the cost associated with a foreseeable risk. Otherwise liability can ensue. In Figure [Figure 1-1](#), Burden is the parabolically increasing line, and risk, or Probability multiplied by Loss, is the parabolically decreasing line. While these lines are not related to a specific measurement, their parabolic shape is meant to reflect the last mile problem in removing all ML system risk, and that the application of additional care beyond a reasonable threshold leads to diminishing returns for decreasing risk as well.

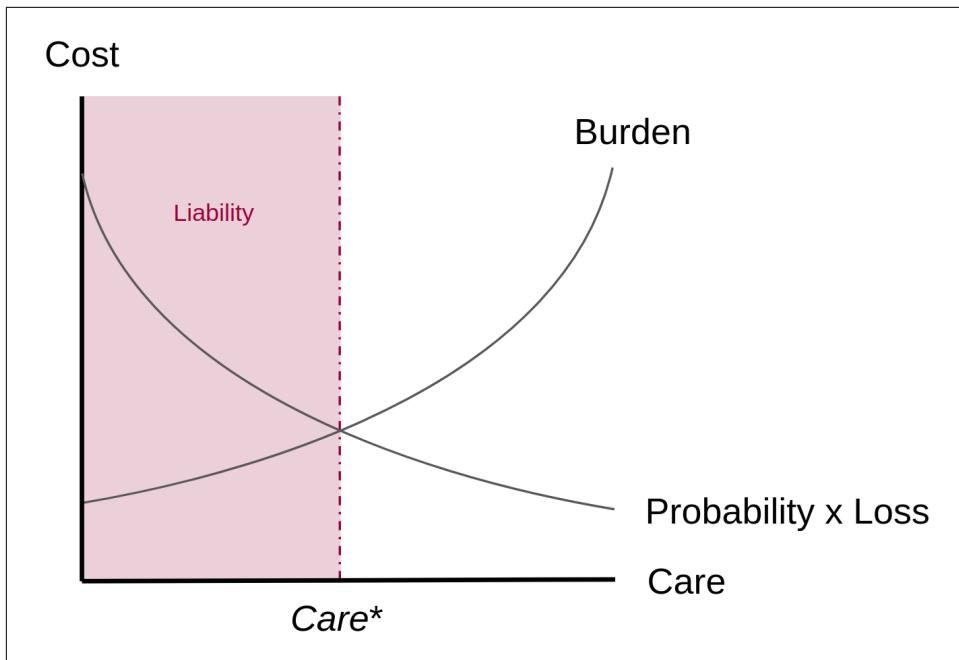


Figure 1-1. An illustration of the Hand Rule. Adapted from [Economic Analysis of Alternative Standards of Liability in Accident Law](#).



A fairly standard definition for the *risk* of a technology incident is the estimated likelihood of the incident multiplied by its estimated cost. More broadly, ISO defines risk in the context of enterprise risk management as the “effect of uncertainty on objectives.”

While it's probably too resource intensive to calculate the quantities in the Hand Rule exactly, it is important to think about these concepts of negligence and liability when designing an ML system. For a given ML system, if the probability of an incident is high, if the monetary, or other, loss associated with a system incident is large, or both quantities are large, organizations need to spend extra resources on ensuring safety for that system. Moreover, organizations should document to the best of their ability that due diligence exceeds the estimated failure probabilities multiplied by the estimated losses.

Federal Trade Commission Enforcement

How might we actually get in trouble? If you're working in a regulated industry, you probably know your regulators. But if we don't know if our work is regulated or who might be enforcing consequences if we cross a legal or regulatory red line, it's proba-

bly the US Federal Trade Commission (FTC) we need to be most concerned with. The FTC is broadly focused on unfair, deceptive or predatory trade practices, and they have found reason to take down at least three prominent ML algorithms in three years. With their new enforcement tool, *algorithmic disgorgement*, the FTC has the ability to delete algorithms and data, and typically, prohibit future revenue generation from an offending algorithm. **Cambridge Analytica** was the first firm to face this new punishment, after their deceptive data collection practices surrounding the 2016 election. **Everalbum** and **WW**, known as Weight Watchers, have also experienced the pain of disgorgement.

The FTC has been anything but quiet about its intention to enforce federal laws around AI and ML. FTC commissioners have pinned lengthy treatises on *Algorithms and Economic Justice*. They have also posted at least two blogs providing high-level guidance for companies who would like to avoid the unpleasantness of enforcement actions. These blogs highlight a number of concrete steps organizations should take. For example, in *Using Artificial Intelligence and Algorithms*, the FTC makes it clear that consumers should not be misled into interacting with an ML system posing as a human. Accountability is another prominent theme in *Using Artificial Intelligence and Algorithms*, in *Aiming for Truth, Fairness, and Equity in Your Company's Use of AI*, and other related publications. In *Aiming for Truth, Fairness, and Equity in Your Company's Use of AI*, the FTC states, "**Hold yourself accountable – or be ready for the FTC to do it for you.**" (Emphasis added by the original author.) This is extremely direct language is unusual from a regulator. In *Using Artificial Intelligence and Algorithms*, the FTC puts forward, "Consider how you hold yourself accountable, and whether it would make sense to use independent standards or independent expertise to step back and take stock of your AI." The next section introduces some of the emerging independent standards we can use to increase accountability, make better products and to decrease any potential legal liability.

Authoritative Best Practices

Data science mostly lacks professional standards and licensing today, but some authoritative guidance is starting to appear on the horizon. The International Standards Organization (ISO) is beginning to outline *technical standards for AI*. Making sure our models are in line with ISO standards would be one way to apply an independent standard to our ML work. Particularly for US-based data scientists, the NIST AI RMF is a very important project to watch.

Version 1 of the AI RMF is intended to be released in January of 2023. The framework puts forward characteristics for trustworthiness in AI systems: validity, reliability, safety, security, resiliency, transparency, accountability, explainability, interpretability, bias management and enhanced privacy. Then it presents actionable guidance across four organizational functions — map, measure, manage, and govern

— for achieving trustworthiness. The guidance in the map, measure, manage, and govern functions is sub-divided into more detailed categories and sub-categories. To see these categories of guidance checkout the [current draft](#) of the RMF or the [AI RMF playbook](#) that provides even more detailed suggestions.



The NIST AI Risk Management framework is a *voluntary* tool for improving the trustworthiness of AI and ML systems. The AI RMF is not regulation and NIST is not a regulator.

To follow our own advice, and that of regulators and publishers of authoritative guidance, and to make this book more useful, we'll be calling out how *we think* the content of each chapter aligns to the AI RMF. Below readers will find a callout box that matches the chapter sub-headings with AI RMF sub-categories. The idea is that readers can use the table to understand how employing the approaches discussed in each chapter may help them adhere to the AI RMF. Because the sub-category advice may sound abstract to ML practitioners in some cases, we hope that being able to match a certain subcategory with our more practice-oriented language will be helpful in translating the RMF into *in vivo* ML deployments. Check out the callout box below to see how we think [Chapter 1](#) matches to the AI RMF, and look for similar tables at the start of each chapter. We've also chosen to use underbars instead of spaces for the designation of AI RMF sub-category names so that readers can look them up in the index of this book.



NIST does not review, approve, condone or otherwise address any content in this book, including any claims relating to the AI RMF. All AI RMF content is simply the authors' opinions and in no way reflects an official position of NIST or any official or unofficial relationship between NIST and the book or the authors.

NIST AI RMF Crosswalk

Chapter Section	NIST AI RMF Sub-categories
A Snapshot of the Legal and Regulatory Landscape	GOVERN_1.1, GOVERN_1.2, GOVERN_2.2, GOVERN_4.1, GOVERN_5.2, MAP_1.1, MAP_1.2, MAP_3.3
Authoritative Best Practices	GOVERN_1.2, GOVERN_5.2, MAP_1.1, MAP_2.3,
AI Incidents	GOVERN_1.2, GOVERN_1.4, GOVERN_4.3, GOVERN_6.2, MANAGE_4.1
Organizational Accountability	GOVERN_1, GOVERN_2, GOVERN_4, GOVERN_5, GOVERN_6.2

Chapter Section	NIST AI RMF Sub-categories
Culture of Effective Challenge	GOVERN_4
Drinking Your Own Champagne	GOVERN_4.1, MEASURE_4.2
Diverse and Experienced Teams	GOVERN_3, MAP_1.2
Moving Fast and Breaking Things	GOVERN_2.1, GOVERN_4.1
Forecasting Failure Modes	GOVERN_4.2, MAP_1.1, MAP_2.3, MAP_3.2, MAP_5.1, MAP_5.3, MANAGE_2
Risk-tiering	GOVERN_5.2, MAP_5.2
Model Documentation	GOVERN_1, GOVERN_2.1, GOVERN_4.2, MAP, MEASURE_1.1, MEASURE_2, MEASURE_3.1, MEASURE_4, MANAGE
Model Monitoring	GOVERN_1.2, GOVERN_1.3, GOVERN_1.4, MAP_5.1, MEASURE_2.5, MEASURE_2.6, MEASURE_2.7, MEASURE_2.9, MEASURE_2.10
Model Inventories	GOVERN_1.3, MAP_4
System Validation and Process Auditing	GOVERN_1.2, GOVERN_1.3, GOVERN_2.1, GOVERN_4.1, GOVERN_4.3, GOVERN_6.1, MAP_2.3, MAP_4, MEASURE
Change Management	GOVERN_1.2, GOVERN_1.3, GOVERN_2.2, GOVERN_4.2, MAP_4.1, MANAGE_4.1
Model Audits and Assessments	GOVERN_2.1, GOVERN_4.1, MAP_4.1, MEASURE
Impact Assessments	GOVERN_2.1, GOVERN_4.2, GOVERN_5.2, MAP_5.1, MAP_5.2, MANAGE_1
Appeal and Override	GOVERN_1.4, GOVERN_5.1, MANAGE_4.1
Pair and Double Programming	GOVERN_4.1, GOVERN_5.2
Security Permissions for Model Deployment	GOVERN_1.4, GOVERN_4.1, GOVERN_5.2, MAP_4.2
Bug Bounties	GOVERN_5.1
AI Incidents Response	GOVERN_1.2, GOVERN_1.4, GOVERN_4.3, GOVERN_6.2, MAP_5.2, MANAGE_4.1

Applicable AI trustworthiness characteristics include: Valid_and_Reliable, Safe, Managed_Bias, Secure_and_Resilient, Transparent_and_Accountable, Explainable_and_Interpretable, Enhanced_Privacy

See also:

- [NIST AI Risk Management Framework](#)
- [NIST AI Risk Management Framework Playbook](#)

AI Incidents

In many ways, the fundamental goal of the ML safety processes and related model debugging discussed in [Chapter 3](#), is to prevent and mitigate AI incidents. Here, we'll loosely define AI incidents as any outcome of the system that could cause harm. And using the Hand rule as a guide, the severity of an AI incident is increased by the loss the incident causes, and decreased by the care taken by the operators to mitigate those losses.

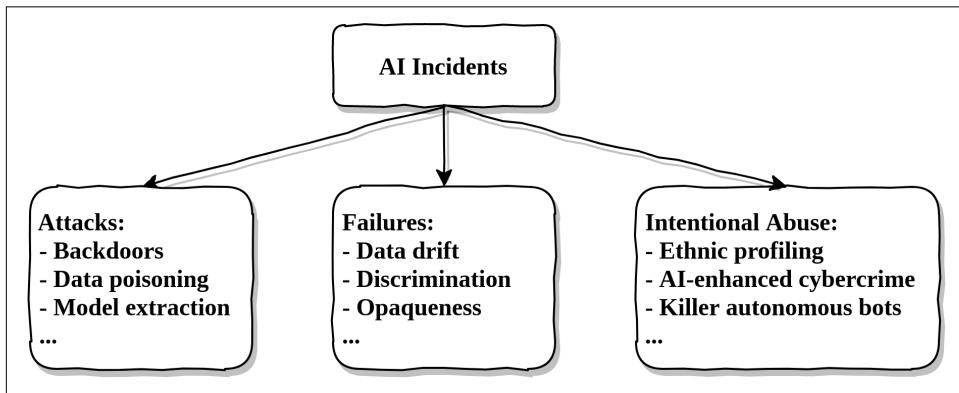


Figure 1-2. A basic taxonomy of AI incidents. Adapted from [What to Do When AI Fails](#).

Because complex systems drift toward failure, there is no shortage of AI incidents to discuss as examples. AI incidents can range from annoying to deadly — from [mall security robots falling down stairs](#), to [self-driving cars killing pedestrians](#), to [mass-scale diversion of healthcare resources](#) away from those who need them most. As pictured in Figure [Figure 1-2](#), AI incidents can be roughly divided into three buckets.

- **Abuses:** AI can be used for nefarious purposes, apart from specific hacks and attacks on AI systems. The day may already be have come where hackers use AI to increase the efficiency and potency of their more general attacks. What the future could hold is even more frightening. Specters like autonomous drone attacks and ethnicity profiling by authoritarian regimes are already on the horizon.
- **Attacks:** Examples of all major types of attacks - confidentiality, integrity, and availability attacks (see [Chapter 5](#) for more information) - have been published by researchers. Confidentiality attacks involve the exfiltration of training data or model logic from AI system end-points. Integrity attacks include adversarial manipulation of training data or model outcomes, either through adversarial examples, evasion, impersonation, or poisoning. Availability attacks can be conducted through more standard denial-of-service approaches, through sponge

examples that overuse system resources, or via algorithmic discrimination induced by some adversary to deny system services to certain groups of users.

- **Failures:** AI system failures tend to involve algorithmic discrimination, safety and performance lapses, data privacy violations, inadequate transparency, or problems in third party system components.

AI incidents are a reality. And like the systems from which they arise, AI incidents can be complex. AI incidents have multiple causes: failures, attacks, and abuses. They also tend to blend traditional notions of computer security, with concerns like data privacy and algorithmic discrimination.

The 2016 [Tay chatbot](#) incident is an informative example. Tay was a state-of-the-art chatbot trained by some of the world's leading experts at Microsoft Research for the purpose of interacting with people on Twitter to increase awareness about AI. Sixteen hours after its release - and 96,000 tweets later - Tay had spiraled into a neo-nazi pornographer and had to be shut down. What happened? Twitter users quickly learned that Tay's adaptive learning system could easily be poisoned. Racist and sexual content tweeted at the bot was incorporated into its training data, and just as quickly resulted in offensive output. Data poisoning is an integrity attack, but due to the context in which it was carried out, this attack resulted in algorithmic discrimination. It's also important to note that Tay's designers, being world-class experts at an extremely well-funded research center, seemed to have put some guide rails in place. Tay would respond to certain hot-button issues with pre-canned responses. But that was not enough, and Tay devolved into a public security and algorithmic discrimination incident for Microsoft Research.

Think this was a one-off incident? Wrong. Just recently, again due to hype and failure to think through performance, safety, privacy and security risks systematically, many of Tay's most obvious failures were [repeated in ScatterLab's release of its Lee Luda chatbot](#). When designing ML systems, plans should be compared to past known incidents in hope of preventing future similar incidents. This is precisely the point of recent [AI incident database](#) efforts and associated [publications](#).

AI incidents can also be an apolitical motivator for responsible technology development. For better or worse, cultural and political viewpoints on topics like algorithmic discrimination and data privacy can vary widely. Getting a team to agree on ethical considerations can be very difficult. It might be easier to get them working to prevent embarrassing and potentially costly or dangerous incidents, which should be a baseline goal of any serious data science team. The notion of AI incidents is central to understanding ML safety and a central theme of this chapter's content is cultural competencies and business processes that can be used to prevent and mitigate AI incidents. We'll dig into those mitigants in the next sections and take a deep dive into a real incident to close the chapter.

Cultural Competencies for Machine Learning Risk Management

An organization's culture is an essential aspect of responsible AI. This section will discuss the cultural competencies like accountability, drinking our own champagne, domain expertise, and the stale adage, "move fast and break things."

Organizational Accountability

A key to the successful mitigation of ML risks is real accountability within organizations for AI incidents. If no one's job is at stake when an ML system fails, gets attacked, or is abused for nefarious purposes, then it's entirely possible that no one in that organization really cares about ML safety and performance. In addition to developers who think through risks, apply software quality assurance (QA) techniques, and model debugging methods, organizations need individuals or teams who validate ML system technology and audit associated processes. Organizations also need someone to be responsible for AI incident response plans. All of this is why leading financial institutions, whose use of predictive modeling has been regulated for decades, employ a practice known as **model risk management** (MRM). MRM is patterned off the Federal Reserve's **S.R. 11-7 model risk management guidance**, that arose out of the Great Recession financial crisis. Notably, implementation of MRM often involves accountable executives and several teams that are responsible for safety and performance of models and ML systems.

Implementation of MRM standards usually requires several different teams and executive leadership. These are some of the key tenets that form the cultural backbone for MRM:

- **Written Policies and Procedures:** The organizational rules for making and using ML should be written and available for all organizational stakeholders. Those close to ML systems should have trainings on the policies and procedures. These rules should also be audited to understand when they need to be updated. No one should be able to claim ignorance of the rules, the rules should be transparent, and the rules should not change without approval. Policies and procedures should include clear mechanisms for escalating serious risks or problems to senior management, and likely put forward whistleblower processes and protections.
- **Effective Challenge:** Effective challenge dictates that experts with the capability to change a system, who did not build the ML system being challenged, perform validation and auditing. MRM practices typically distribute effective challenge across three "lines of defense," where conscientious system developers make up the first line of defense and independent, skilled and empowered technical validators and process auditors make up the second and third lines, respectively.

- **Accountable Leadership:** A specific executive within an organization should be accountable for ensuring AI incidents do not happen. This position is often referred to as *chief model risk officer* (CMRO). It's also not uncommon for CMRO terms of employment and compensation to be linked to ML system performance. The role of CMRO offers a very straightforward cultural check on ML safety and performance. If our boss really cares about ML system safety and performance, then we start to care too.
- **Incentives:** Data science staff and management must be incentivized to implement ML responsibly. Often, compressed product timelines can incentivize the creation of a minimum viable product first, with rigorous testing and remediation relegated to the end of the model life cycle immediately before deployment to production. Moreover, ML testing and validation teams are often evaluated by the same criterion as ML development teams, leading to a fundamental misalignment where testers and validators are encouraged to move quickly rather than assure quality. Aligning timeline, performance evaluation, and pay incentives to team function helps solidify a culture of responsible ML and risk mitigation.

Of course, small or young organizations may not be able to spare an entire full-time employee to monitor ML system risk. But it's important to have an individual or group held accountable if ML systems cause incidents and rewarded if the systems work well. If an organization assumes everyone is accountable for ML risk and AI incidents, the reality is that no one is accountable.

Culture of Effective Challenge

Whether our organization is ready to adopt full-blown MRM practices, or not, we can still benefit from certain aspects of MRM. In particular, the cultural competency of effective challenge can be applied outside of the MRM context. At its core, effective challenge means actively challenging and questioning steps in the development of ML systems. An organizational culture that encourages serious questioning of ML system designs will be more likely to develop effective ML systems or products, and to catch problems before they explode into harmful incidents. Note that effective challenge cannot be abusive, and it must apply equally to all personnel developing an ML system, especially so-called “rockstar” engineers and data scientists. Effective challenge should also be structured, such as weekly meetings where current design thinking is questioned and alternative design choices are seriously considered.

Diverse and Experienced Teams

Diverse teams can bring wider and uncorrelated perspectives to bear on design, development, and testing ML systems. Non-diverse teams often do not. Many have documented the unfortunate outcomes that can arise as a result of data scientists not considering demographic diversity in the training or results of ML systems. A poten-

tial solution to these kinds of oversights is increasing demographic diversity on ML teams from its **current woeful levels**. Business or other domain experience is also important when building teams. Domain experts are instrumental in feature selection and engineering, and in the testing of system outputs. In the mad rush to develop ML systems, domain expert participation can also serve as a safety check. Generalist data scientists often lack the experience necessary to deal with domain-specific data and results. Misunderstanding the meaning of input data or output results is a recipe for disaster that can lead to AI incidents when a system is deployed. Unfortunately, the social sciences deserve a special emphasis when it comes to data scientists forgetting or ignoring the importance of domain expertise. In a trend referred to as "**tech's quiet colonization of the social sciences**," several organizations have pursued regrettable ML projects that seek to **replace decisions that should be made by trained social scientists** or that simply **ignore the collective wisdom of social science domain expertise** altogether.

Drinking Our Own Champagne

Also known as “eating our own dog food,” the practice of drinking our own champagne refers to using our own software or products inside of our own organization. Often a form of pre-alpha or pre-beta testing, drinking our own champagne can identify problems that emerge from the complexity of *in vivo* deployment environments before bugs and failures affect customers, users or the general public. Because serious issues like concept drift, algorithmic discrimination, shortcut learning or underspecification are notoriously difficult to identify using standard ML development processes, drinking our own champagne provides a limited and controlled, but also realistic, test bed for ML systems. Of course, when organizations employ demographically and professionally diverse teams, including domain experts in the field where the ML system will be deployed, drinking our own champagne is more likely to catch a wider variety of problems. Drinking our own champagne also brings the classical Golden Rule into AI. If we’re not comfortable using a system on ourselves or in our own organization, then we probably shouldn’t deploy that system.

Moving Fast and Breaking Things

The mantra, “move fast and break things,” is almost a religious belief for many “rock-star” engineers and data scientists. Sadly, these top practitioners also seem to forget that when they go fast and break things, things get broken. As ML systems make more high-impact decisions that implicate autonomous vehicles, credit, employment, grades and university attendance, medical diagnoses and resource allocation, mortgages, pre-trial bail, parole and more, breaking things means more than buggy apps. It can mean that a small group of data scientists and engineers causes real harm at scale to many people. Participating in the design and implementation of high-impact ML systems requires a mindset change to prevent egregious performance and safety

problems. Practitioners must change from prioritizing the number of software features they can push or the test data accuracy of an ML model, to recognizing the implications and downstream risks of their work.

Organizational Processes for Machine Learning Risk Management

Organizational processes play a key role in assuring ML systems are safe and performant. Like the cultural competencies discussed in the previous section, organizational processes are a key non-technical determinant of reliability in ML systems. This section on processes starts out by urging practitioners to consider, document, and attempt to mitigate any known or foreseeable failure modes for their ML systems. We then discuss more about MRM. While the culture section focused on the people and mindsets necessary to make MRM a success, this section will outline the different processes MRM uses to mitigate risks in advanced predictive modeling and ML systems. While MRM is an incredible process standard to which we can all aspire, there are additional important process controls that are not typically part of MRM. We'll look beyond traditional MRM in this section and highlight crucial risk control processes like pair or double programming and security permission requirements for code deployment. This section will close with a discussion of AI incident response. No matter how hard we work to minimize harms while designing and implementing an ML system, we still have to prepare for failures and attacks.

Forecasting Failure Modes

ML safety and ethics experts roughly agree on the importance of thinking through, documenting, and attempting to mitigate foreseeable failure modes for ML systems. Unfortunately they also mostly agree that this is a nontrivial task. Happily, new resources and scholarship on this topic have emerged in recent years that can help ML system designers forecast incidents in more systematic ways. If holistic categories of potential failures can be identified, it makes hardening ML systems for better real-world performance and safety a more pro-active and efficient task. In this subsection, we'll discuss one such strategy, along with a few additional processes for brainstorming future incidents in ML systems.

Known Past Failures

As discussed in *Preventing Repeated Real World AI Failures by Cataloging Incidents: The AI Incident Database*, one of the most efficient ways to mitigate potential AI incidents in our ML systems is to compare our system design to past failed designs. Much like transportation professionals investigating incidents, cataloging incidents, using the findings to prevent related incidents, and to test new technologies, several ML researchers, commentators, and trade organizations have begun to collect and ana-

lyze AI incidents in hopes of preventing repeated and related failures. Likely the most high-profile and mature AI incident repository is the [AI Incident Database](#). This searchable and interactive resource allows registered users to search a visual database with keywords and locate different types of information about publicly recorded incidents.

Consult this resource while developing ML systems. If a system similar to the one we're currently designing, implementing, or deploying has caused an incident in the past, this is one of strongest indicators that our new system could cause an incident. If we see something that looks familiar in the database, we should stop and think about what we're doing a lot more carefully.

Failures of Imagination

Imagining the future with context and detail is never easy. And it's often the context in which ML systems operate, accompanied by unforeseen or unknowable details, that lead to AI incidents. In a recent workshop paper, the authors of [*Overcoming Failures of Imagination in AI Infused System Development and Deployment*](#) put forward some structured approaches to hypothesize about those hard-to-imagine future risks. In addition to deliberating on the *who* (e.g., investors, customers, vulnerable non-users), *what* (e.g., well-being, opportunities, dignity), *when* (e.g., immediately, frequently, over long periods of time), and *how* (e.g., taking an action, altering beliefs) of AI incidents, they also urge system designers to consider:

- Assumptions that the impact of the system will be only beneficial, and to admit when uncertainty in system impacts exists.
- The problem domain and applied use cases of the system, as opposed to just the math and technology.
- Any unexpected or surprising results, user interactions, and responses to the system.

Causing AI incidents is embarrassing, if not costly or illegal, for organizations. AI incidents can also hurt consumers and the general public. Yet, with some foresight, many of the currently known AI incidents could have been mitigated, if not wholly avoided. It's also possible that in performing the due diligence of researching and conceptualizing ML failures, we find that our design or system must be completely reworked. If this is the case, take comfort that a delay in system implementation or deployment is likely less costly than the harms our organization or the public could experience if the flawed system was released.

Model Risk Management Processes

The process aspects of MRM mandate thorough documentation of modeling systems, human review of systems, and ongoing monitoring of systems. These processes rep-

resent the bulk of the governance burden for the Federal Reserve's SR 11-7 MRM guidance, which is overseen by the Federal Reserve and the Office of the Comptroller of the Currency (OCC) for predictive models deployed in material consumer finance applications. While only large organizations will be able to fully embrace all that MRM has to offer, any serious ML practitioner can learn something from the discipline. The subsection below breaks MRM processes down into smaller components so that readers can start thinking through using aspects of MRM in their organization.

Risk-tiering

As outlined in the opening of [Chapter 1](#), the product of the probability of a harm occurring and likely loss resulting from that harm is an accepted way to rate the risk of given ML system deployment. The product of risk and loss has a more formal name in the context of MRM, *materiality*. Materiality is a powerful concept that enables organizations to assign realistic risk levels to ML systems. More importantly, this risk-tiering allows for the efficient allocation of limited development, validation, and audit resources. Of course, the highest materiality applications should receive the greatest human attention and review, while the lowest materiality applications could potentially be handled by automatic machine learning (AutoML) systems and undergo minimal validation. Because risk mitigation for ML systems is an ongoing expensive task, proper resource allocation between high, medium, and low risk systems is a must for effective governance.

Model Documentation

MRM standards also require that systems be thoroughly documented. First, documentation should enable accountability for system stakeholders, ongoing system maintenance, and a degree of incident response. Second, documentation must be standardized across systems for the most efficient audit and review processes. Documentation is where the rubber hits the road for compliance. Documentation templates, illustrated at a very high level by the section list below, are documents that data scientists and engineers fill in as they move through a standardized workflow or in the later stages of model development. Documentation templates should include all the steps that a responsible practitioner should conduct to build a sound model. If parts of the document aren't filled out, that points to sloppiness in the training process. Since most documentation templates and frameworks also call for adding one's name and contact information to the finished model document, there should be no mystery about who is not pulling their weight. For reference, the section list below is a rough combination of typical sections in MRM documentation and the sections recommended by the [Annexes to the EU Artificial Intelligence Act](#).

- Basic Information

- Names of Developers and Stakeholders
- Current Date and Revision Table
- Summary of Model System
- Business or Value Justification
- Intended Uses and Users
- Potential Harms and Ethical Considerations
- Development Data Information
 - Source for Development Data
 - Data Dictionary
 - Privacy Impact Assessment
 - Assumptions and Limitations
 - Software Implementation for Data Preprocessing
- Model Information
 - Description of Training Algorithm with Peer-reviewed References
 - Specification of Model
 - Performance Quality
 - Assumptions and Limitations
 - Software Implementation for Training Algorithm
- Testing Information
 - Quality Testing and Remediation
 - Discrimination Testing and Remediation
 - Security Testing and Remediation
 - Assumptions and Limitations
 - Software Implementation for Testing
- Deployment Information
 - Monitoring Plans and Mechanisms
 - Up- and Down-stream Dependencies
 - Appeal and Override Plans and Mechanisms
 - Audit Plans and Mechanisms
 - Change Management Plans
 - Incident Response Plans

- References (If we're doing science, then we're building on the shoulders of giants and we'll have several peer-reviewed references in a formatted bibliography!)

Of course, these documents can be hundreds of pages long, especially for high-materiality systems. The proposed **datasheet** and **model card** standards may also be helpful for smaller or younger organizations to meet these goals. If readers are feeling like lengthy model documentation sounds impossible for their organization today, then maybe these two simpler frameworks might work instead.

Model Monitoring

A primary tenant of ML safety is that ML system performance in the real-world is hard to predict and performance must be monitored. Hence, deployed system performance should be monitored frequently and until a system is decommissioned. Systems can be monitored for any number of problematic conditions, the most common being input drift. While ML system training data encodes information about a system's operating environment in a static snapshot, the world is anything but static. Competitors can enter markets, new regulations can be promulgated, consumer tastes can change, and pandemics or other disasters can happen. Any of these can change the live data that's entering our ML system away from the characteristics of its training data, resulting in decreased, or even dangerous, system performance. To avoid such unpleasant surprises, the best ML systems are monitored both for drifting input and output distributions and for decaying quality, often known as *model decay*. While performance quality is the most common quantity to monitor, ML systems can also be monitored for anomalous inputs or predictions, specific attacks and hacks, and for drifting fairness characteristics.

Model Inventories

Any organization that is deploying ML should be able to answer straightforward questions like:

- How many ML systems are currently deployed?
- How many customers or users do these systems affect?
- Who are the accountable stakeholders for each system?

MRM achieves this goal through the use of model inventories. A model inventory is a curated and up-to-date database of all an organization's ML systems. Model inventories can serve as a repository for crucial information in documentation, but should also link to monitoring plans and results, auditing plans and results, important past and upcoming system maintenance and changes, and plans for incident response.

System Validation and Process Auditing

Under traditional MRM practices, an ML system undergoes two primary reviews before its release. The first review is a technical validation of the system, where skilled validators, not uncommonly Ph.D. level data scientists, attempt to poke holes in system design and implementation, and work with system developers to fix any discovered problems. The second review investigates processes. Audit and compliance personnel carefully analyze the system design, development, and deployment, along with documentation and future plans, to ensure all regulatory and internal process requirements are met. Moreover, because ML systems change and drift over time, review must take place whenever a system undergoes a major update or at an agreed upon future cadence.

Readers may be thinking (again) that their organization doesn't have the resources for such heavy-handed reviews. Of course that is a reality for many small or younger organizations. The keys for validation and auditing, that should work at nearly any organization, are having technicians who did not develop the system test it, having a function to review non-technical internal and external obligations, and having sign-off oversight for important ML system deployments.

Change Management

Like all complex software applications, ML systems tend to have a large number of different components. From backend ML code, to application programming interfaces (APIs), to graphical user interfaces (GUIs), changes in any component of the system can cause side-effects in other components. Add in issues like data drift, emergent data privacy and anti-discrimination regulations, and complex dependencies on third-party software, and change management in ML systems becomes a serious concern. If we're in the planning or design phase of a mission-critical ML system, we'll likely need to make change management a first-class process control. Without explicit planning and resources for change management, process or technical mistakes that arise through the evolution of the system, like using data without consent or API mismatches, are very difficult to prevent. Furthermore, without change management, such problems might not even be detected until they cause an incident.

We'll circle back to MRM throughout the book. It's one of the most battle-tested frameworks for governance and risk management of ML systems. Of course, MRM is not the only place to draw inspiration for improved ML safety and performance processes, and the next subsection will draw out lessons from other practice areas.



Reading the 21-page [SR 11-7 model risk management guidance](#) is a quick way to up-skill yourself in ML risk management. When reading it, pay special attention to the focus on cultural and organizational structures. Managing technology risks is often more about people than anything else.

Beyond Model Risk Management

There are many ML risk management lessons to be learned from financial audit, data privacy, software development best practices and from IT security. This subsection will shine a light on ideas that exist outside the purview of traditional MRM: model audits, impact assessments, appeal, override, and opt-out, pair or double programming, least privilege, bug bounties and incident response from an ML safety and performance perspective.

Model Audits and Assessments

Audit is a common term in MRM, but it also has meanings beyond what is typically the third line of defense in a more traditional MRM scenario. The phrase *model audit* has come to prominence in recent years, and it seems to mean an official testing and transparency exercise focusing on an ML system that tracks adherence to some policy, regulation, or law. Model audits tend to be conducted by independent third parties with limited interaction between auditor and auditee organizations. For a good break down of model audits, check out the recent paper *Algorithmic Bias and Risk Assessments: Lessons from Practice*. The [paper](#) quoted at the beginning of the chapter puts forward a solid framework for audits and assessments, even including worked documentation examples. The related term, *model assessment*, seems to mean a more informal and cooperative testing and transparency exercise that may be undertaken by external or internal groups.

ML audits and assessments may focus on bias issues or other serious risks including safety, data privacy harms, and security vulnerabilities. Whatever their focus, audits and auditors have to be fair and transparent. Those conducting audits should be held to some ethical or professional standards, which barely exist as of today. Without these kinds of accountability mechanisms or binding guidelines, audits can be an ineffective risk management practice, and worse, tech-washing exercises that certify harmful ML systems. Despite flaws, audits are an en vogue favorite risk control of policy makers and researchers, and are being written into laws. For example, New York City local law 144 discussed above.

Impact Assessments

Impact assessments are a formal documentation approach used in many fields to forecast and record the potential issues a system could cause once implemented. Likely due to their use in [data privacy](#), impact assessments are starting to show up in organizational ML policies and [proposed laws](#). Impact assessments are an effective way to think through and document the harms that an ML system could cause, increasing accountability for designers and operators of AI systems. But impact assessments are not enough on their own. Remembering the definition of risk and materiality put forward above, impact is just one factor in risk. Impacts must be com-

bined with likelihoods to form a risk measure, then risks must be actively mitigated, where the highest risk applications tend to receive the most oversight. Impact assessments are just the beginning of a broader risk management process. Like other risk management processes, they must be performed at a cadence that aligns to the system being assessed. If a system changes quickly, it will need more frequent impact assessments. Another potential issue with impact assessments is when they are designed and implemented by the ML teams that are also being assessed. In this case, there will be a temptation to diminish the scope of the assessment and downplay any potential negative impacts. Impact assessments are an important part of broader risk management and governance strategies, but they must be conducted as often as required by a specific system, and likely conducted by independent oversight professionals.

Appeal, Override and Opt-out

Ways for users or operators to appeal and override inevitable wrong decisions should be built into most ML systems. It's known by many names across disciplines: actionable recourse, intervenability, redress, or adverse action notices. This can be as simple as the "Report inappropriate predictions" function in the Google search bar, or it can be as sophisticated as presenting data and explanations to users and enabling appeal processes for demonstrably wrong data points or decision mechanisms. Another similar approach, known as opt-out, is to let users do business with an organization the old-fashioned way without going through any automated processing. Many data privacy laws and major US consumer finance laws address recourse, opt-out or both. Automatically forcing wrong decisions on many users is one of the clearest ethical wrongs in ML. We shouldn't fall into an ethical, legal, and reputational trap that's so clear and so well-known, but many systems do. That's likely because it takes planning and resources for both processes and technology, laid out from the beginning of designing an ML system, to get appeal, override, and opt-out right.

Pair and Double Programming

Because they tend to be complex and stochastic, it's hard to know if any given ML algorithm implementation is correct! This is why some leading ML organizations implement ML algorithms twice as a quality assurance (QA) mechanism. Such double implementation is usually achieved by one of two methods: pair programming or double programming. In the pair programming approach, two technical experts code an algorithm without collaborating. Then they join forces and work out any discrepancies between their implementations. In double programming, the same practitioner implements the same algorithm twice, but in very different programming languages, such as Python (object-oriented) and SAS (procedural). They must then reconcile any differences between their two implementations. Either approach tends to catch numerous bugs that would otherwise go unnoticed until the system was deployed. Pair and double programming can also align with the more standard workflow of data scientists prototyping algorithms, while dedicated engineers harden them for

deployment. However, for this to work, engineers must be free to challenge and test data science prototypes and not relegated to simply re-coding prototypes.

Security Permissions for Model Deployment

The concept of *least privilege* from IT security states that no system user should ever have more permissions than they need. Least privilege is a fundamental process control that, likely because ML systems touch so many other IT systems, tends to be thrown out the window for ML build-outs and for so-called “rock star” data scientists. Unfortunately, this is an ML safety and performance anti-pattern. Outside the world of over-hyped ML and rock star data science, it’s long been understood that engineers cannot adequately test their own code and that others in a product organization, product managers, attorneys, or executives, should make the final call as to when software is released.

For these reasons, the IT permissions necessary to deploy an ML system should be distributed across several teams within an IT organizations. During development sprints, data scientists and engineers certainly must retain full control over their development environments. But, as important releases or reviews approach, the IT permissions to push fixes, enhancements, or new features to user-facing products are transferred away from data scientists and engineers to product managers, testers, legal, executives or others. Such process controls provide a gate that prevents unapproved code from being deployed.

Bug Bounties

Bug bounties are another concept we can borrow from computer security. Traditionally a bug bounty is when an organization offers rewards for finding problems in its software, particularly security vulnerabilities. Since ML is mostly just software, we can do bug bounties for ML systems. While we can use bug bounties to find security problems in ML systems, we can also use them to find other types of problems related to reliability, safety, transparency, explainability, interpretability, or privacy. Through bug bounties, we use monetary rewards to incentivize community feedback in a standardized process. As we’ve highlighted elsewhere in the chapter, incentives are crucial in risk management. Generally, risk management work is tedious and resource consuming. If we want our users to find major problems in our ML systems for us, we need to pay them or reward them in some other meaningful way. Bug bounties are typically public endeavors. If that makes some organizations nervous, internal hackathons in which different teams look for bugs in ML systems may have some of the same positive effects. Of course, the more participants are incentivized to participate, the better the results are likely to be.

AI Incident Response

According to the vaunted [SR 11-7 guidance](#), “even with skilled modeling and robust validation, model risk cannot be eliminated”. If risks from ML systems and ML models cannot be eliminated, then such risks will eventually lead to incidents. Incident response is already a mature practice in the field of computer security. Venerable institutions like [NIST](#) and [SANS](#) have published computer security incident response guidelines for years. Given that ML is a less mature and higher-risk technology than general purpose enterprise computing, formal AI incident response plans and practices are a must for high-impact or mission critical AI systems.

Formal AI incident response plans enable organizations to respond more quickly and effectively to inevitable incidents. Incident response also plays into the Hand Rule discussed at the beginning of the chapter. With rehearsed incident response plans in place, organizations may be able to identify, contain, and eradicate AI incidents before they spiral into costly or dangerous public spectacles. AI incident response plans are one of the most basic and universal ways to mitigate AI-related risks. Before a system is deployed, incident response plans should be drafted and tested. For young or small organizations that cannot fully implement model risk management, AI incident response is a cost-effective and potent AI risk control to consider. Borrowing from computer incident response, AI incident response can be thought of in six phases:

Phase 1: Preparation. In addition to clearly defining an AI incident for our organization, preparation for AI incidents includes personnel, logistical, and technology plans for when an incident occurs. Budget must be set aside for response, communication strategies must be put in place, and technical safeguards for standardizing and preserving model documentation, out-of-band communications, and shutting down AI systems must be implemented. One of the best ways to prepare and rehearse for AI incidents are table top discussion exercises, where key organizational personnel work through a realistic incident. Good starter questions for an AI incident table top include:

- Who has the organizational budget and authority to respond to an AI incident?
- Can the AI system in question be taken offline? By whom? At what cost? What upstream processes will be affected?
- Which regulators or law enforcement agencies need to be contacted? Who will contact them?
- Which external law firms, insurance agencies, or public relation firms need to be contacted? Who will contact them?
- Who will manage communications? Internally, between responders? Externally, with customers or users?

Phase 2: Identification. Identification is when organizations spot AI failures, attacks, or abuses. Identification also means staying vigilant for AI-related abuses. In practice, this tends to involve more general attack identification approaches, like network intrusion monitoring, and more specialized monitoring for AI system failures, like monitoring for concept drift or algorithmic discrimination. Often the last step of the identification phase is to notify management, incident responders, and others specified in incident response plans.

Phase 3: Containment. Containment refers to mitigating the incident's immediate harms. Keep in mind that harms are rarely limited to the system where the incident began. Like more general computer incidents, AI incidents can have network effects that spread throughout an organization's and its customers' technologies. Actual containment strategies will vary depending on whether the incident stemmed from an external adversary, and internal failure, or an off-label use or abuse of an AI system. If necessary, containment is also a good place to start communicating with the public.

Phase 4: Eradication. Eradication involves remediating any affected systems. For example, sealing off any attacked systems from vectors of in- or ex-filtration, or shutting down a discriminatory AI system and temporarily replacing it with a trusted rule-based system. After eradication, there should be no new harms caused by the incident.

Phase 5: Recovery. Recovery means ensuring all affected systems are back to normal and that controls are in place to prevent similar incidents in the future. Recovery often means re-training or re-implementing AI systems, and testing that they are performing at documented pre-incident levels. Recovery can also require careful analysis of technical or security protocols for personnel, especially in the case of an accidental failure or insider attack.

Phase 6: Lessons Learned. *Lessons learned* refers to corrections or improvements of AI incident response plans based on the successes and challenges encountered while responding to the current incident. Response plan improvements can be process- or technology-oriented.

When reading the case below, think about the phases of incident response, and whether AI incident response would have been an effective risk control for Zillow.

Case Study: The Rise and Fall of Zillow's iBuying

In 2018, the real estate tech company Zillow entered the business of buying homes and flipping them for a profit, known as *iBuying*. The company believed that its proprietary, ML-powered *Zestimate* algorithm could do more than draw eyeballs to its extremely popular web products. As reported by Bloomberg, Zillow employed

domain experts to validate the numbers generated by their algorithms when they first started to buy homes. First, local real estate agents would price the property. The numbers were combined with the Zestimate, and a final team of experts vetted each offer before it was made.

According to [Bloomberg](#), Zillow soon phased out these teams of domain experts in order to, “get offers out faster,” preferring the speed and scale of a more purely-algorithmic approach. When the Zestimate did not adapt to a rapidly-inflating real estate market in early 2021, Zillow reportedly intervened to increase the attractiveness of its offers. As a result of these changes, the company began acquiring properties at a rate of nearly 10,000 homes per quarter. More flips means more staff and more renovation contractors, but as Bloomberg puts it, “Zillow’s humans couldn’t keep up.” Despite increasing staffing levels by 45% and bringing on “armies” of contractors, the iBuying system was not achieving profitability. The combination of pandemic staffing and supply challenges, the over-heated housing market, and complexities around handling large numbers of mortgages were just too much for the iBuying project to manage.

In October of 2021, Zillow announced that it would stop making offers through the end of the year. As a result of Zillow’s appetite for rapid growth, as well as labor and supply shortages, the company had a huge inventory of homes to clear. To solve its inventory problem, Zillow was posting most homes for resale at a loss. Finally, on November 2, Zillow announced that it was writing down its inventory by over 500 million dollars. Zillow’s foray into the automated house-flipping business was over.

Fallout

In addition to the huge monetary loss of its failed venture, Zillow announced that it would lay off about 2,000 employees - a full quarter of the company. In June of 2021, Zillow was trading at around \$120 per share. At the time of this writing nearly one year later, shares are approximately 40 dollars, erasing over \$30 billion in stock value. (Of course, the entire price drop can’t be attributed to the iBuying incident, but it certainly factored into the loss.) The downfall of Zillow’s iBuying is rooted in many interweaving causes, and cannot be decoupled from the pandemic that struck in 2020 and upended the housing market. Below, we’ll examine how to apply what we’ve learned in this chapter about governance and risk management to Zillow’s misadventures.

Lessons Learned

What does [Chapter 1](#) teach us about the Zillow iBuying saga? Based on the public reporting, it appears Zillow’s decision to sideline human-review of high-materiality algorithm’s was probably a factor in the overall incident. We also question whether Zillow had adequately thought through the financial risk it was onboarding, whether

appropriate governance structures were in place, and whether the iBuying losses might have been handled better as an AI incident. We don't know the answer for many of these questions with respect to Zillow, so instead we'll focus on learnings readers can apply at their own organization.

- **Lesson 1: Validate with domain experts:** In this chapter, we stressed the importance of diverse and experienced teams as a core organizational competency for responsible ML development. Without a doubt, Zillow has internal and external access to world-class expertise in real estate markets. However, in the interest of speed and automation — sometimes referred to as “moving fast and breaking things” or “product velocity” — Zillow phased the experts out of the process of acquiring homes, choosing instead to rely on their Zestimate algorithm. According to follow up reporting by [Bloomberg](#) in May 2022, “Zillow told its pricing experts to stop questioning the algorithms, according to people familiar with the process.” This choice may have proved fatal for the venture, especially in a rapidly changing, pandemic-driven real estate market. No matter the hype, AI is not smarter than humans yet. If we’re making high-risk decisions with ML, keep humans in the loop.
- **Lesson 2: Forecast failure modes:** The coronavirus pandemic of 2020 created a paradigm shift in many domains and markets. ML models, that usually assume that the future will resemble the past, likely suffered across the board in many verticals. We shouldn’t expect a company like Zillow to see a pandemic on the horizon. But as we discuss above, rigorously interrogating the failure modes of our ML system constitutes a crucial competency for ML in high-risk settings. We do not know the details of Zillow’s model governance frameworks, but the downfall of Zillow’s iBuying stresses the importance of effective challenge and asking hard questions, like “what happens if the cost of performing renovations doubles over the next two years?” and “what will be the business cost of overpaying for homes by two percent over the course of six months?” For such a high-risk system, probable failure modes should be enumerated and documented, likely with board of directors oversight, and the actual financial risk should have been made clear to all senior decision-makers. At our organization, we need to know the cost of being wrong with ML and that senior leadership is willing to tolerate those costs. Maybe senior leaders at Zillow were accurately informed of iBuying’s financial risks, maybe they weren’t. What we know now is that Zillow took a huge risk, and it did not pay off.
- **Lesson 3: Governance counts:** Zillow’s CEO is famous for risk-taking, and has a proven track record of winning big bets. But, we simply can’t win every bet we make. This is why we manage and govern risks when conducting automated decision-making, especially in high-risk scenarios. SR 11-7 states, “the rigor and sophistication of validation should be commensurate with the bank’s overall use of models.” Zillow is not a bank, but Bloomberg’s May 2022 post-mortem puts it

this way. Zillow was “attempting to pivot from selling online advertising to operating what amounted to a hedge fund and a sprawling construction business.” Zillow drastically increased the materiality of their algorithms, but appears not to have drastically increased governance over those algorithms. As we noted above, most the public reporting points to Zillow decreasing human oversight of their algorithms during their iBuying program, not increasing oversight. A separate risk function, empowered with the organizational stature to stop models from moving into production, appropriate budget and staff levels, that reports directly to the board directors, and operates independently from business and technology functions headed by the CEO and CTO, is common in major consumer finance organization. This organizational structure, when it works as intended, allows for more objective and risk-based decisions about ML model performance, and avoids the conflicts of interest and confirmation bias that tends to occur when business and technology leaders evaluate their own systems for risk. We don’t know if Zillow had an independent model governance function — it is quite rare these days outside of consumer finance. But we do know, no risk or oversight function was able to stop the iBuying program before losses became staggering. While it’s a tough battle to fight as a single technician, helping our organization apply independent audits to its ML systems is a workable risk mitigation practices for ML systems.

- **Lesson 4: AI incidents occur at scale:** Zillow’s iBuying hijinks aren’t funny. Money was lost. Careers were lost — thousands of employees were laid off or resigned. This looks like a \$30 billion AI incident. From the incident response lens, we need to be prepared for systems to fail, we need to be monitoring for systems to fail, and we need to have documented and rehearsed plans in place for containment, eradication, and recovery. From public reporting, it does appear that Zillow was aware of their iBuying problems, but their culture was more focused on winning big than preparing for failure. Given the size of the financial loss, Zillow’s containment efforts could have been more effective. Zillow was able to eradicate their most acute problems with the declaration of the roughly half-billion dollar write-off in November of 2021. As for recovery, Zillow’s leadership has plans for a new real estate super app, but given the stock price today, recovery is a long way off and investors are weary. Complex system’s drift toward failure. Perhaps a more disciplined incident-handling approach could save our organization when it bets big with ML.

The final and most important lesson we can take away from the Zillow Offers saga is at the heart of this book. Emerging technologies always come with risks. Early automobiles were dangerous. Planes used to crash into mountainsides much more frequently. ML systems can perpetuate discriminatory practices, pose security and privacy risks, and behave unexpectedly. A fundamental difference between ML and other emerging technologies is that these systems can make decisions quickly and at

huge scales. When Zillow leaned into its Zestimate algorithm, it could scale up its purchasing to hundreds of homes per day. In this case, the result was a write-down of half of a billion dollars, even larger stock losses, and the loss of thousands of jobs. This phenomenon of rapid failure at scale can be even more directly devastating when the target of interest is access to capital, social welfare programs, or the decision of who gets a new kidney.

Resources

- ISO standards for AI
- NIST AI Risk Management Framework
- *Supervisory Guidance on Model Risk Management*

Interpretable and Explainable Machine Learning

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcrönin@oreilly.com.

Scientists have been fitting models to data to learn more about observed patterns for centuries. Explainable machine learning (ML) models and post-hoc explanation of ML models present an incremental, but important, advance in this long-standing practice. Because ML models learn about nonlinear, faint, and interacting signals more easily than traditional linear models, humans using explainable ML models and post-hoc explanation techniques can now also learn about nonlinear, faint, and interacting signals in their data with more ease.

In [Chapter 2](#) we’ll dig into important ideas for interpretation and explanation before tackling major explainable modeling and post-hoc explanation techniques. We’ll cover the major pratfalls of post-hoc explanation too — many of which can be overcome by using explainable models and post-hoc explanation *together*. Next we’ll discuss applications of explainable models and post-hoc explanation, like model

documentation and actionable recourse for wrong decisions, that increase accountability for AI systems. Chapter 2 will close with a case discussion of the so called “A-Levels scandal” in the United Kingdom (UK) where an explainable, highly documented model made unaccountable decisions, resulting in a nationwide AI incident.



NIST does not review, approve, condone or otherwise address any content in this book, including any claims relating to the AI RMF. All AI RMF content is simply the authors' opinions and in no way reflects an official position of NIST or any official or unofficial relationship between NIST and the book or the authors.

NIST AI RMF Crosswalk

Chapter Section	NIST AI RMF Sub-categories
Important Ideas for Interpretability and Explainability	GOVERN_1.1, GOVERN_1.2, GOVERN_1.4, GOVERN_5.1, GOVERN_6.1, MAP_2.3, MAP_3.1, MEASURE_2.8
Explainable Models	GOVERN_1.1, GOVERN_1.2, GOVERN_6.1, MAP_2.3, MAP_3.1, MEASURE_2.8
Post-hoc Explanation	GOVERN_1.1, GOVERN_1.2, GOVERN_1.4, GOVERN_5.1, GOVERN_6.1, MAP_2.3, MAP_3.1, MEASURE_2.1, MEASURE_2.8

Applicable AI trustworthiness characteristics include: Valid_and_Reliable, Secure_and_Resilient, Transparent_and_Accountable, Explainable_and_Interpretable

See also:

- *Four Principles of Explainable Artificial Intelligence*
- *Psychological Foundations of Explainability and Interpretability in Artificial Intelligence*

Important Ideas for Interpretability and Explainability

Before jumping into the techniques for training explainable models and generating post-hoc explanations, we need to discuss the big ideas behind the math and code. We'll start by affirming that transparency does not equate to trust. We can trust things that we don't understand, and understand things we don't trust. Stated even more simply: transparency enables understanding, and understanding is different than trust. In fact, greater understanding of poorly built ML systems may actually decrease trust.

Trustworthiness in AI is defined by the National Institute of Standards and Technology (NIST) using various characteristics: validity, reliability, safety, managed bias, security, resiliency, transparency, accountability, explainability, interpretability, and enhanced privacy. Transparency makes it easier to achieve other desirable trustworthiness characteristics and makes debugging easier. However, human operators must take these additional governance steps. Trustworthiness is often achieved in practice through testing, monitoring, and appeal processes (see [Chapter 1](#) and [Chapter 3](#)). Increased transparency enabled by explainable ML models and post-hoc explanation should enable the kinds of diagnostics and debugging that help to make traditional linear models trustworthy. It also means that regulated applications that have relied on linear models for decades, due to per-consumer explanation and general documentation requirements, are now likely ripe for disruption with accurate and transparent ML models.



Being interpretable, explainable, or otherwise transparent does not make a model good or trustworthy. Being interpretable, explainable, or otherwise transparent enables humans to make a highly informed decision as to whether a model is good or trustworthy.

Redress for subjects of wrong ML-decisions via prescribed appeal and override processes is perhaps the most important trust-enhancing use of explainable models and post-hoc explanation. The ability to logically appeal automated decisions is sometimes called *actionable recourse*. It's very hard for consumers — or job applicants, patients, prisoners, or students — to appeal automated black-box decisions. Explainable ML models and post-hoc explanation techniques should allow for ML-based automated decisions to be understood by decision subjects, which is the first step in a logical appeal process. Once users can demonstrate that either their input data or the logic of their decision is wrong, operators of ML systems should be able to override the initial errant decision.



Mechanisms that enable appeal and override of automated decisions should always be deployed with high-risk ML systems.

It's also important to differentiate between interpretability and explanation. In the groundbreaking study, *Psychological Foundations of Explainability and Interpretability in Artificial Intelligence*, researchers at NIST were able to differentiate between interpretability and explainability using widely-accepted notions of human cognition. According to NIST researchers, the similar but not identical concepts of interpretation and explanation are defined as follows:

- **Interpretation:** a high-level, meaningful mental representation that contextualizes a stimulus and leverages human background knowledge. An interpretable model should provide users with a description of what a data point or model output means *in context*.
- **Explanation:** a low-level, detailed mental representation that seeks to describe some complex process. An ML explanation is a description of how some model mechanism or output *came to be*.



Interpretability is a much higher bar to reach than explainability. Achieving interpretability means putting some ML mechanism or result in context, and this is rarely achieved through models or post-hoc explanations. Interpretability is usually achieved through plainly written explanations, compelling visualizations or interactive graphical user interfaces. Interpretability usually requires working with domain, user interaction/user experience experts, and other interdisciplinary professionals.

Let's get more granular now, and touch on some of what makes an ML model interpretable or explainable. It's very difficult for models achieve transparency if their input data is a mess. Let's start our discussion there. When considering input data and transparency, think about the following:

- **Explainable feature engineering:** We should avoid overly complex feature engineering if our goal is transparency. While deep features from autoencoders, principal components, or high-degree interactions might make our model perform better in test data, explaining such features is going to be difficult even if we feed them into an otherwise explainable model.
- **Meaningful features:** Using a feature as an input to some model function assumes it is related to the function output, i.e., the model's predictions. Using nonsensical or loosely related features because they improve test data performance violates a fundamental assumption of the way explaining models works. For example, if along with other features we use eye color to predict credit default, the model will likely train to some convergence criterion and we can calculate Shapley additive explanation values (SHAP) for eye color. But eye color has no real validity in this context and is not causally related to credit default. While eye color may proxy for underlying systemic biases, claiming that eye color explains credit default is invalid. Apply common sense, over even better — causal discovery approaches, to increase the validity of models and associated explanations.
- **Monotonic features:** Monotonicity helps with explainability. While noise in input features may prevent them from appearing totally monotonic, we can apply

an interpretable feature engineering technique known as binning or discretization to place features into monotonically increasing or decreasing buckets prior to training explainable models.

If we can rely on our data being usable for explainability and interpretability purposes, then we can use concepts like additive independence of inputs, constraints, linearity and smoothness, prototypes, sparsity and summarization to ensure our model is as transparent as possible.

- **Additivity of Inputs:** Keeping inputs in an ML model separate, or only interacting in a small group, is crucial for transparency. Traditional ML models are (in)famous for combining and re-combining input features into an undecipherable tangle of high-degree interactions. In stark contrast, traditional linear models treat inputs in an independent and additive fashion. The output decision from a linear model is typically the simple linear combination of learned model parameters and input feature values. Of course, performance quality is often noticeably worse for traditional linear models as compared traditional black-box ML models.

Enter the generalized additive model (GAM). GAMs keep input features independent, enabling transparency, but also allow for arbitrarily complex modeling of each feature's behavior, dramatically increasing performance quality. We'll also be discussing GAM's close descendants, GA2Ms and explainable boosting machines (EBM) in the next section. They all work by keeping inputs independent and enabling visualization of the sometimes complex manner in which they treat those individual inputs. In the end, they retain high transparency because users don't have to disentangle inputs and how they affect one another, and the output decision is still a linear combination of learned model parameters and some function applied to data inputs values.

- **Constraints:** Traditional black-box ML models are revered for their flexibility. They can model almost any signal-generating function in training data. Yet, when it comes to transparency, modeling every nook and cranny of some observed response function usually isn't a great idea. Due to overfitting, turns out it's also not a great idea for performance on unseen data either. Sometimes what we observe in training data is just noise, or even good-old-fashioned wrong. So, instead of overfitting to bad data, it's often a good idea to apply constraints that both increase transparency and help with performance on unseen data by forcing models to obey causal concepts instead of noise.

Any number of constraints can be applied when training ML models, but some of the most helpful and widely-available are sparsity, monotonicity and interaction constraints. Sparsity constraints, often implemented by L1 regularization (which usually decreases the number of parameters or rules in a model), increases emphasis on a more manageable number of input parameters and internal learn-

ing mechanisms. Positive monotonic constraints mean that as a model input increases, its output must never decrease. Negative monotonic constraints ensure that as a model input increases, its output can never increase. Interaction constraints keep internal mechanisms of ML models from combining and recombining too many different features. These constraints can be used to encourage models to learn interpretable and explainable causal phenomenon, as opposed focusing on non-robust inputs, arbitrary nonlinearities, and high-degree interactions that may be drivers of errors and bias in model outcomes.

- **Linearity and smoothness:** Linear functions are monotonic by default and can be described by single numeric coefficients. Smooth functions are almost always differentiable, meaning they can be summarized at any location with a derivative function or value. Basically linear and smooth functions are better behaved and typically easier to summarize. In contrast unconstrained and arbitrary ML functions can bounce around or bend themselves in knots in ways that defy human understanding, making the inherent summarization that needs to occur for explanation nearly impossible.
- **Prototypes:** Prototypes refer to well-understood data points (rows) or archetypal features (columns) that can be used to explain model outputs for some other previously unseen data. Prototypes appear in many places in ML, and have been used to explain and interpret model decisions for decades. Think of explaining a k -nearest neighbors prediction using the nearest neighbors or profiling clusters based on centroid locations — those are prototypes. Prototypes also end up being important for counterfactual explanations. Prototypes have even entered into the typically complex and opaque world of computer vision with [this-looks-like-that deep learning](#).
- **Sparsity:** ML models, for better or worse, can now be trained with [trillions of parameters](#). Yet, it's debatable whether their human operators can reason based on more than a few dozen of those parameters. The volume of information in a contemporary ML model will have to be summarized be transparent. Generally, the less coefficients or rules in an ML model, the more sparse and explainable it is.
- **Summarization:** Summarization can take many forms, including variable importance measures, surrogate models, and other post-hoc ML approaches. Visualization is perhaps the most common vehicle for communicating summarized information about ML models, and approximation taken to compress information is the [Achille's heel](#) for summarization. Additive, linear, smooth or sparse models are generally easier to summarize, and post-hoc explanation processes have a better chance of working well in these cases.

Achieving transparency with ML takes some extra elbow grease as compared to popular and commoditized black-box approaches. Fear not. The recent paper [Designing](#)

Inherently Interpretable Machine Learning Models and software packages like `interpret`, `h2o` and `PiML` provide a great framework for training and explaining transparent models, and the remaining sections of the chapter highlight the most effective technical approaches, and common gotchas, for us to consider next time we start to design an AI system.

Explainable Models

For decades, many ML researchers and practitioners alike labored under a seemingly logical assumption that more complex models were more accurate. However, as pointed out by luminary professor Cynthia Rudin, in her impactful *Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead*, “it is a myth that there is necessarily a trade-off between accuracy and interpretability.” We’ll dig into the tension around explainable models versus post-hoc explanation later in the chapter. For now, let’s concentrate on the powerful idea of accurate and explainable models. They offer the potential for highly accurate decision-making coupled with improved human learning from machine learning, actionable recourse processes and regulatory compliance, improved security, and better debug-ability for accuracy, bias, and other types of issues. These appealing characteristics make explainable ML models a general win-win for practitioners and consumers.

We’re going to go over some of the most popular types of explainable ML models below. We’ll start with the large class of additive models, including penalized regression, GAMs, GA2Ms and EBMs. We’ll also be covering decision trees, constrained tree ensembles, tree-based models and a litany of other options before moving onto post-hoc explanation techniques.

Additive Models

Perhaps the most widely used types of explainable ML models are those based on traditional linear models: penalized regression, GAMs, and GA2Ms (or EBMs). These techniques use contemporary methods to augment traditional modeling approaches, often yielding noticeable improvements in performance. But they also keep interpretability high by treating input features in a separate and additive manner, or by allowing only a small number of interaction terms. They also rely on straightforward visualization techniques to enhance interpretability.

Penalized Regression

We’ll start our discussion of explainable models with penalized regression. Penalized regression updates the typical 19th Century approach to regression for the 21st Century. These types of models usually produce linear, monotonic response functions with globally explainable results like those of traditional linear models, but often with

a boost in predictive performance. Penalized regression models eschew the assumption-laden **Normal Equation** approaches to finding model parameters, and instead use more sophisticated constrained and iterative optimization procedures that allow for handling of correlation, feature selection, and treatment of outliers, all while using validation data to pick a better model automatically.

In Figure [Figure 2-1](#) below, we can see how a penalized regression model learns optimal coefficients for six input features over more than 80 training iterations. We can see at the beginning of the optimization procedure that all parameter values start very small, and grow as the procedure begins to converge. This happens because it's typical to start the training procedure with large penalties on input features. These penalties are typically decreased as training proceeds to allow a small number of inputs to enter the model, to keep model parameters artificially small, or both. (Readers may also hear penalized regression coefficients referred to as "shrunken" on occasion.) At each iteration, as more features enter the model, or as coefficient values grow or change, the current model is applied to validation data. Training continues to a pre-defined number of iterations or until performance in validation data ceases to improve.

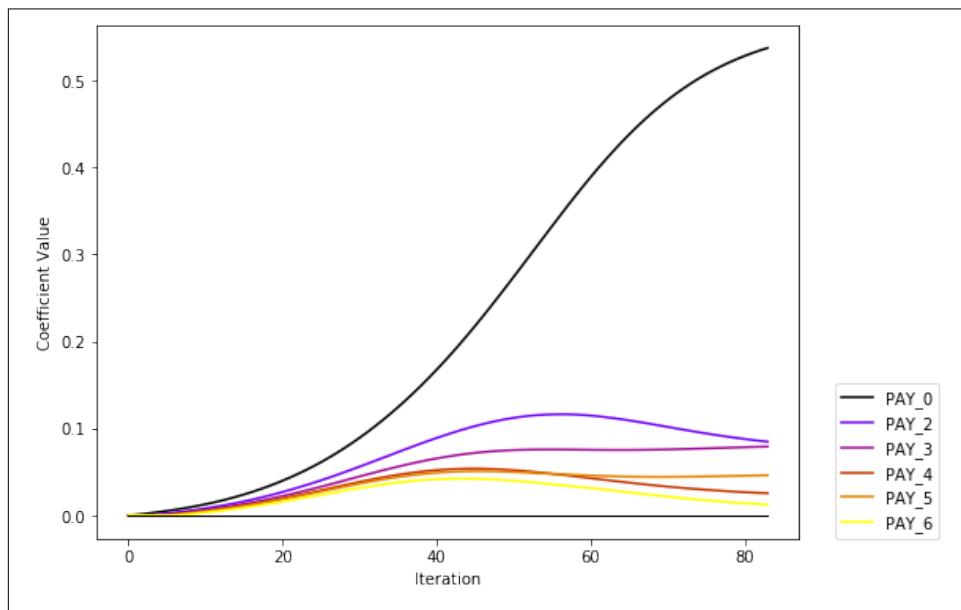


Figure 2-1. Regularization paths for selected features in an elastic net regression model.

In addition to a validation-based early stopping procedure, borrowed directly from the best ML training procedures, penalized regression often uses two types of penalties, on the L1 and L2 parameter norms, iteratively reweighted least squares (IRLS), and link functions to address feature selection, correlated inputs, outliers, and nonlinearity respectively.

- **IRLS:** IRLS is a well-established procedure for minimizing the effect of outliers. It starts like an old-fashioned regression, but after that first iteration, the IRLS procedure checks which rows of input data are leading to large errors. It then reduces the weight of those rows in subsequent iterations of fitting model coefficients. IRLS continues this fitting and down-weighting procedure until model parameters converge to stable values.
- **L1 Norm Penalty:** Also known as least absolute shrinkage and selection operator (LASSO), L1 penalties keep the sum of the absolute model parameters to a minimum. This penalty has the effect of driving unnecessary regression parameters to zero, and selecting a small, representative subset of features for the regression model, while also avoiding potential multiple comparison problems that arise in older stepwise feature selection. When used alone, L1 penalties are known to increase performance quality in situations where a large number of potentially correlated input features cause stepwise feature selection to fail.
- **L2 Norm Penalty:** Also known as ridge or Tikhonov regression, L2 penalties minimize the sum of squared model parameters. L2 norm penalties stabilize model parameters, especially in the presence of correlation. Unlike L1 norm penalties, L2 norm penalties do not select features. Instead, they limit each feature's impact on the overall model by keeping all model parameters smaller than they would be in the traditional solution. Smaller parameters make it harder for any one feature to dominate a model and for correlation to cause strange behavior during model training.
- **Link Functions:** Link functions enable linear models to handle common distributions of training data, such as using a logit link function to fit a logistic regression to input data with two discrete outcomes. Other common and useful link functions include the Poisson link function for count data or an inverse link function for gamma-distributed outputs. Matching outcomes to their distribution family and link function, such as a binomial distribution and logit link function for logistic regression, is absolutely necessary for training deployable models. Many ML models and libraries outside of penalized regression packages often do not support the necessary link functions and distribution families to address fundamental assumptions in training data.

Contemporary penalized regression techniques usually combine the following in a technique known as elastic net:

- Validation-based early stopping for improved generalization
- IRLS to handle outliers
- L1 penalties for feature selection purposes
- L2 penalties for robustness

- Link functions for various target or error distributions

Readers can learn more about penalized regression in *Elements of Statistical Learning*, but for our purposes, it's more important to know when we might want to try penalized regression. Penalized regression has been applied widely across many research disciplines, but it is a great fit for business data with many features, even data sets with more features than rows, and for data sets with a lot of correlated variables. Penalized regression models also preserve the basic interpretability of traditional linear models, so think of them when we have many correlated features or need maximum transparency. It's also important to know penalized regression techniques don't always create confidence intervals, t -statistics, or p -values for regression parameters. These types of measures are typically only available through bootstrapping that can require extra computing time. The R packages `elasticnet` and `glmnet` are maintained by the inventors of the LASSO and elastic net regression techniques, and the `h2o generalized linear model` sticks closely to the implementation of the original software, while allowing for much-improved scalability.

An even newer and extremely interesting twist to penalized regression models is the super-sparse linear integer model, or **SLIM**. SLIMs also rely on sophisticated optimization routines, with the objective of creating accurate models that only require simple arithmetic to evaluate. SLIMs are meant to train linear models that can be evaluated mentally by humans working in high-risk settings, such as healthcare. If we're ever faced with an application that requires the highest interpretability and the need for field-workers to evaluate results quickly, think of SLIMs. On the other hand, if we're looking for better predictive performance, to match that of many black-box ML techniques, but to keep a high-degree interpretability, think of GAMs — which we'll be covering next.

Generalized Additive Models

GAMs are a generalization of linear models that allow a coefficient and function to be fit to each model input, instead of just a coefficient for each input. Training models in this manner allows for each input variable to be treated in a separate but nonlinear fashion. Treating each input separately keeps interpretability high. Allowing for non-linearity improves performance quality. Traditionally GAMs have relied on splines to fit nonlinear shape functions for each input, and most implementations of GAMs generate convenient plots of the fitted shape functions. Depending on our regulatory or internal documentation requirements, we may be able to use the shape functions directly in predictive models for increased performance. If not, we may be able to eyeball some fitted shape functions and switch them out for a more explainable polynomial, log, trigonometric or other simple function of the input feature that may also increase prediction quality. An interesting twist to GAMs was introduced recently with **neural additive models** (NAMs) and **GAMI-Nets**. In these models, an artificial neural network is used to fit the shape functions. We'll continue the theme of estimat-

ing shape functions with ML below when we discuss explainable boosting machines. The Rudin group also recently put forward a variant of GAMs in which monotonic step functions are used as shape functions for maximum interpretability. Check out to *Fast Sparse Classification for Generalized Linear and Additive Models* to see those in action.



We'll use the term *shape function* to describe the learned non-linear relationship that GAM-like models apply to each input and interaction feature. These shape functions may be traditional splines, or they can be fit by machine learning estimators like boosted trees or neural networks.

The semi-official name for the ability to change out parts of a model to better match reality or human intuition is *model editing*. Being editable is another important aspect of many explainable models. Models often learn wrong or biased concepts from wrong or biased training data. Explainable models enable human users to spot bugs, and edit them out. The GAM family of models is particularly amenable to model editing which is another advantage of these powerful modeling approaches. Readers can learn more about GAMs in *Elements of Statistical Learning*. To try GAMs, look into the R `gam` package or the more experimental `h2o` or `pygam` implementations.

GA2M and Explainable Boosting Machines

GA2M and EBM represent straightforward and material improvements to GAMs. Let's address GA2M first. The "2" in GA2M refers to the consideration of a small group of pairwise interactions as inputs to the model. Choosing to include a small number of interaction terms in the GAM boosts performance, and again, without compromising interpretability. Interaction terms can be plotted as contours along with the standard two-dimensional input feature plots that tend to accompany GAMs. Some astute readers may already be familiar with EBM, an important variant of GA2M. In an EBM, instead of fitting the response function for each input feature with spline shape functions, each shape function is fit with boosted trees. Boosted tree shape functions present some conveniences over traditional spline shape functions, in that they tend to be more scalable and can more easily accept categorical or missing data as training inputs. Because of these advances, GA2Ms and EBMs now rival, or exceed, black-box ML models for performance quality, while also presenting the obvious advantages of interpretability and model editing. If our next project is on structured data, let's try out EBMs using the `interpret` package from Microsoft Research. EBMs put the final touches on our discussion of additive models. We'll move on next to decision trees, yet another type of high-quality, high-interpretability model with a track record of success across statistics, data mining, and ML.

Decision Trees

Decision trees are another type of popular predictive model. When used as single trees, and not as ensemble models, they learn highly-interpretable flowcharts from training data and tend to exhibit better predictive performance than linear models. When used as ensembles, like random forests and GBMs, they lose interpretability but tend to even better predictors. Below, we'll discuss both single tree models and constrained decision tree ensembles that can retain some level of explainability.

Single Decision Trees

Technically, decision trees are directed graphs in which each interior node corresponds to an input feature. There are graph edges to child nodes for values of the input feature that creates the highest target purity, or increased predictive quality, in each child node. Each terminal node or leaf node represents a value of the target feature given the values of the input features represented by the path from the root to the leaf. These paths can be visualized or explained with simple if-then rules.

In plainer language, decision trees are data-derived flowcharts, just like we see in Figure [Figure 2-2](#) below. Decision trees are great for training interpretable models on structured data. They are beneficial when the goal is to understand relationships between the input and target variable with Boolean-like “if-then” logic. They are also great at finding interactions in training data. Parent-child relationships, especially near the top of the tree, tend to point toward feature interactions that can be used to better understand drivers of the modeling target, or they can be used as interaction terms to boost predictive accuracy for additive models. Perhaps their main advantage over more straightforward additive models is their ability to train directly on character values, missing data, non-standardized data, and non-normalized data. In this age of big (err, bad) data, decision trees enable building models with minimal data preprocessing, which can help eliminate additional sources of human bias from ML models.

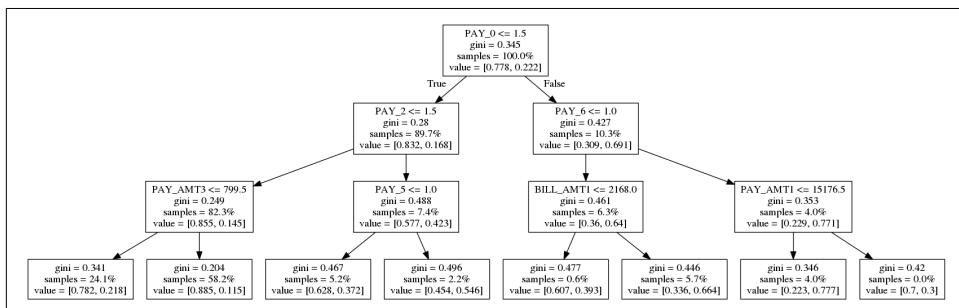


Figure 2-2. A graph representing a simple decision tree model forms a data-derived flowchart.

With all this going for them, why shouldn't we use decision trees? First of all, they're only interpretable when they're shallow. Decision trees with more than, say, five levels of if-then branches become difficult to interpret. They also tend to perform relatively poorly on unstructured data like sound, images, video, and text. Unstructured data has become the domain of deep learning and neural networks. Like many more complex models, decision trees require a high degree of tuning. Decision trees have many hyperparameters, or settings, which must be specified using human domain knowledge, grid-searches, or other hyperparameter tuning methods. Such methods are time-consuming at a minimum, and at worst, sources of bias and overfitting. Single decision trees are also unstable, in that adding a few rows of data to training or validation data and retraining can lead to a completely re-arranged flowchart. Such instability is an Achilles' heel for many ML models.

Decision tree's make locally optimal, or *greedy*, decisions each time they make a new branch or if-then rule. Other ML models use different optimization strategies, but the end result is the same: a model that's not the best model for the dataset, but instead is one good candidate for the best model out of many, many possible options. This issue of many possible models for any given the datasets has at least two names, *the multiplicity of good models* or *the Rashomon effect*. (The name *Rashomon* is from a famous film in which witnesses describe the same murder differently.) The Rashomon effect is linked to another nasty problem, known as *underspecification*, in which hyperparameter tuning and validation-data-based model selection leads to a model that looks good in test scenarios but then flops in the real-world. How do we avoid these problems with decision trees? Mainly by training single trees that we can see, examine, and ensure their logic will hold up when they are deployed.

There are many variations on the broader theme of decision trees. For example, tree-based models fit a linear model in each terminal node of a decision tree, adding predictive capacity to standard decision trees while keeping all the explainability intact. Professor Rudin's research group has introduced *optimal sparse decision trees* as a response to the issues of instability and underspecification. Another answer to these problems are manual constraints, informed by human domain knowledge. We'll address constrained decision tree ensembles next, but if readers are ready to try standard decision trees there are many packages to try, and the R package *rpart* is one the best.

Constrained XGBoost Models

The popular gradient boosting package XGBoost now supports both *monotonicity constraints* and *interaction constraints*. As described earlier, user-supplied monotonicity constraints force the XGBoost model to preserve more explainable monotonic relationships between model inputs and model predictions. The interaction constraints can prevent XGBoost from endlessly recombining features. These software features turn what would normally be a black-box tree ensemble model replete with

underspecification problems into a highly robust model capable of learning from data and accepting causal constraints from expert human users. These newer training options, that enable human domain experts to use causal knowledge to set the direction of modeled relationships and to specify which input features should not interact, combined with XGBoost's proven track record of scalability and performance, make this a hard choice to overlook when it comes to explainable ML. While not as directly interpretable as EBMs, because GBMs combine and recombine features into a tangle of nested if-then rules, constrained XGBoost models are amendable to a wide variety of post-hoc explainable and visualization techniques. The explanation techniques often enable practitioners to confirm the causal knowledge they've injected into the model and to understand the inner-workings of the complex ensemble.

An Ecosystem of Explainable Machine Learning Models

Beyond additive and tree-based models, there is an entire ecosystem of explainable ML models. Some of these models have been known for decades, some represent tweaks to older approaches, and some are wholly new. Whether they're new or old, they challenge the status quo of black-box ML — and that's a good thing. If readers are surprised about all the different options for explainable models, imagine a customer's surprise when we can explain the drivers of our AI system's decisions, help them confirm or deny hypotheses about their business, and provide explanations to users, all while achieving high levels of predictive quality. Indeed, there are lots of options for explainable models, and one that might suit our next project well. Instead of asking our colleagues, customers, or business partners to blindly trust a black box, consider explainable neural networks, rule-based models, k-nearest-neighbors, causal or graphical models, or even sparse matrix factorization next time.

- **Causal Models:** Causal models, in which causal phenomena are linked to some prediction outcome of interest in a provable manner, are often seen as the gold-standard of interpretability. Given we can often look at a chart that defines how the model works and they are composed using human domain or causal inference techniques, they're almost automatically interpretable. They also don't fit to noise in training data as badly as traditional black-box ML models. The only hard part of causal models is finding the data necessary to train them or the training processes themselves. However, training capabilities for causal models continue to improve, and packages like [pymc3](#) for Bayesian models and [dowhy](#) for causal inference have been enabling practitioners to build and train causal models for years. More recently, both Microsoft and Uber released a [tutorial](#) for causal inference with real-world use cases using modeling libraries from both companies, [EconML](#) and [causalml](#) respectively. If we care about stable, interpretable models watch this space carefully. Once considered to be nearly impossible to train, causal models are slowly working their way into the mainstream, and their inher-

ent advantages could make unstable, overfit, black-box ML models obsolete one day.

- **Explainable Neural Networks:** Released and refined by Wells Fargo Bank risk management, explainable neural networks (XNNs) prove that with a little ingenuity and elbow grease, even the most unexplainable black-box models can be made explainable and retain high degrees of performance quality. Explainable neural networks use the same principals as GAMs, GA2Ms, and EBMs to achieve both high interpretability and high predictive performance, but with some twists. Like GAMs, XNNs are simply additive combinations of shape functions. However, they add an additional indexed-structure to GAMs. XNNs are an example of a generalized additive *index* models (GAIMs), where GAM-like shape functions are fed by a lower projection layer that can learn interesting higher-degree interactions.

In an XNN, back-propagation is used to learn optimal combinations of variables (see Figure 4c) to act as inputs into shape functions learned via subnetworks (see Figure 4b), which are then combined in an additive fashion with optimal weights to form the output of the network (see Figure 4a). While likely not the simplest explainable model, XNNs have been refined to be more scalable, they automatically identify important interactions in training data, and post-hoc explanations can be used to break down their predictions into locally-accurate feature contributions. Another interesting type of transparent neural networks was put forward recently in *Neural Additive Models: Interpretable Machine Learning with Neural Nets*. NAMs appear similar to XNNs, except they forego the bottom layer that attempts to locate interaction terms.

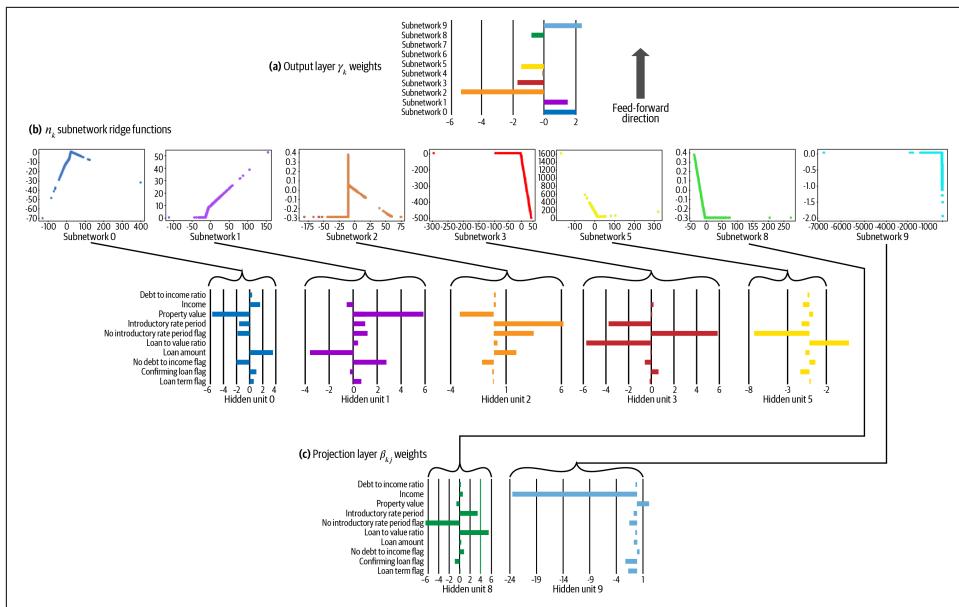


Figure 2-3. A chart describing the innerworkings of an explainable neural network (XNN).

- **k-nearest-neighbors (k-NN):** k-NN uses prototypes, or similar data points, to make predictions. Reasoning in this way does not require training and it's simplicity often appeals to human users. If k is set to 3, an inference on a new point involves finding the 3 nearest points to the new point and taking the average or modal label from those three points as the predicted outcome for the new point. This kind of logic is common in our day-to-day lives. Take the example of residential real-estate appraisal. Price per square foot for one home is often evaluated as the average price for per square foot of three comparable houses. When we say, “it sounds like,” or “it feels like,” or “it looks like,” we’re probably using prototype data points to make inferences about things in our own life. The notion of interpretation by comparison to prototypes is what lead the Rudin group to take on the black-box computer vision with **this-looks-like-that**, a new type of deep learning that uses comparisons to prototypes to explain image classification predictions.
- **Rule-based Models:** Extracting predictive if-then rules from datasets is another long-running type of ML modeling. The simple Boolean logic of if-then rules is interpretable, as long as the number of rules, the number of branches in the rules, and the number of entities in the rules are constrained. **RuleFit** and **scope rules** are two popular techniques that seek to find predictive and explainable rules from training data. Rule-based models are also the domain of the Rudin group.

Among their greatest hits for interpretable and high-quality rule-based predictors are **certifiable optimal rule lists (CORELS)** and **scalable Bayesian rule lists**. Code for these and other valuable contributions from the Rudin group are available on their public [Code](#) page.

- **Sparse Matrix Factorization:** Factoring a large data matrix into two smaller matrices is a common dimension reduction and unsupervised learning technique. Most older matrix factorization techniques re-attribute the original columns of data across dozens of derived features, rendering the results of these techniques unexplainable. However, with the introduction of L1 penalties into matrix factorization, it's now possible to extract new features from a large matrix of data, where only a few of the original columns have large weights on any new feature. By using **sparse principal components analysis (SPCA)**, one might find that when extracting a new feature from customer financial data, that new feature is composed exclusive of the debt-to-income and revolving account balance features in my original dataset. We could then reason through this feature being related to consumer debt. Or if we find another new feature that has high weights for income, payments, and disbursements then we could interpret that feature as relating to cash flow. **Nonnegative matrix factorization (NMF)**, gives similar results but assumes that training data only takes on positive values. For unstructured data like term counts and pixel intensities, that assumption always holds. Hence, NMF can be used to find explainable summaries of topics in documents or to decompose images into explainable dictionaries of sub-components. Whether we use SPCA or NMF, the resulting extracted features can be used as explainable summaries, archetypes for explainable comparisons, axes for visualization, or features in models. And it turns out that just like many explainable supervised learning models are special instances of GAMs, many unsupervised learning techniques are instances of **generalized low-rank models**, which we can try out in h2o.

Now that we've covered the basics of explainable models we'll move onto post-hoc explanation techniques. But before we do, remember that it's perfectly fine to use explainable models and post-hoc explanation together. Explainable models are often used to incorporate domain knowledge into learning mechanisms, to address inherent assumptions or limitations in training data, or to build functional forms that humans have a chance of understanding. Post-hoc explanation is often used for visualization and summarization. While post-hoc explanation is often discussed in terms of increasing transparency for traditional black-box ML models, there's lots of reasons to question that application that we'll introduce below. Using explainable models and post-hoc explanation together, to improve upon and validate the other, may be the best general application for both technologies.

Post-hoc Explanation

We'll start off by addressing global and local feature importance measures, then move onto surrogate models, popular types of plots for describing model behavior, and touch on a few post-hoc explanation techniques for unsupervised learning. We'll also discuss the shortcomings of post-hoc explanations, which can perhaps be broadly summarized in three points below that readers should keep in mind as they form their own impressions and practices with explanations:

1. If a model doesn't make sense to people, its explanations won't either. (We can't explain nonsense.)
2. ML models can easily grow so complex they can't be summarized accurately.
3. It's difficult to convey explanatory information about ML systems to a broad group of users and stakeholders.

Despite these difficulties, post-hoc explanation is almost always necessary for interpretability and transparency. Even models like logistic regression, thought of as highly transparent, must be summarized to meet regulatory obligations around transparency. For better or worse, we're probably stuck with post-hoc explanation and summarization. So let's try to make it work as well as possible.



Many models must be summarized in a post-hoc fashion to be interpretable. Yet, ML explanations are often incorrect, and checking them requires rigorous testing and comparisons to an underlying explainable model.

Feature Importance

Feature importance is one of the most central aspects of explaining ML models. There are many methods for calculating feature importance and the methods tell us how much an input feature contributed to the predictions of a model, either globally — across an entire dataset — or locally — for one or a few rows of data. While a handful of more established global feature importance metrics do not arise from the aggregation of local measures, averaging (or otherwise aggregating) local measures into global measures is common today. We'll start by discussing newer methods for local feature importance below and then move onto global methods.



Global explanations summarize a model mechanism or prediction over an entire dataset or large sample of data. *Local* explanations perform the same type of summarization, but for smaller segments of data, down to a single row or cell of data.

There's also another caveat to keep mind for the feature importance section. "Importance" can be misleading as a name. We're just measuring proxies for importance. Consider, for example, a computer vision security system that relies on gradient-based methods to detect "important" aspects of video frames. Without proper training and deployment specifications, such systems will have a hard time picking up individuals wearing camouflage, as newer digital camouflage clothing is specifically designed to blend into various backgrounds, and to keep visual gradients between the fabric and different backgrounds smooth and undetectable. But isn't people wearing camouflage one of the most important things to detect in a security application? For the rest of this section, remember to consider exactly what we're calculating when we calculate importance for the best real-world results.

Local Explanations and Feature Importance

Determining which input features impacted a specific prediction is crucial to ML explanations. Local feature importance is perhaps the most common way we make this determination, and the phrase typically refers to the raw numeric values that show us how much a feature contributed to a single prediction. Have a look at Figure [Figure 2-4](#) below to see local feature importance in action.

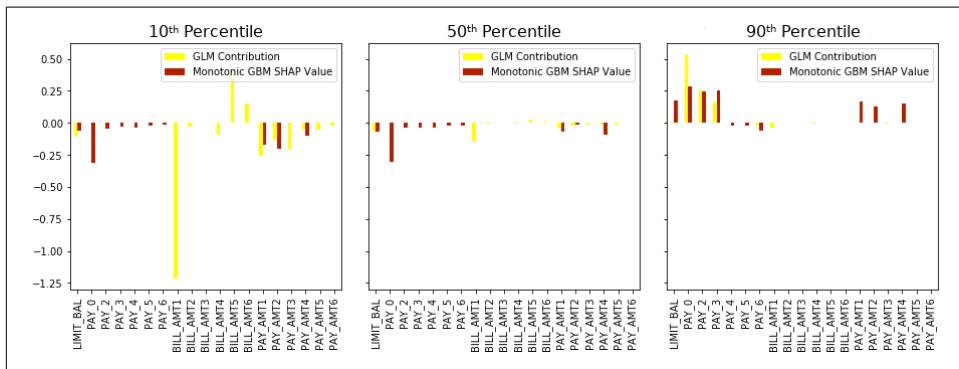


Figure 2-4. Local feature importance for two explainable models for three individuals at the 10th, 50th, and 90th percentiles of predicted probabilities.

Figure [Figure 2-4](#) shows two types of local feature importance values, for two different models, for three different customers in a credit lending example. The first customer sits at the 10th percentile of probability of default, and they are likely to receive the credit product on offer. The second customer sits at the 50th percentile, and are unlikely to receive the product on offer. The third customer sits at the 90th percentile of probability of default and provides an example of an extremely high-risk applicant. The two models being summarized are a penalized GLM and a monotonically constrained GBM. Since both models have relatively simple structures, that make sense in general for the problem at hand, we've done a good job handling the caveats

brought up in our opening remarks about post-hoc explanation. These models roughly make sense, so the explanations should as well. And they are simple enough to be accurately summarized.



One practical way to make explanations more interpretable is to compare them to some meaningful benchmark. That's why we compare our feature importance values to Pearson correlation, and our partial dependence and ICE values to mean model predictions. This should enable viewers of the plot to compare more abstract explanatory values to a more understandable benchmark value.

How are we summarizing their local behavior? For the penalized GLM, we're multiplying the values for the standardized input feature by standardized coefficients for each applicant. For the GBM, we're applying SHAP. Both of these local feature importance techniques are additive and locally accurate — meaning they sum to the model prediction. Both are measured from an offset, the GLM intercept and the SHAP intercept, which are similar in value, but not equal. (That is not accounted for in Figure [Figure 2-4](#).) And both values can be generated in the same space: either log-odds or predicted probability depending on how the calculation is performed. What do the plotted values tell us? For the low-risk applicant at the 10th percentile of probability of default, we can see that their most recent bill amount, `BILL_AMT1`, is highly favorable and driving their prediction downward. The SHAP values for the same customer tell a slightly different story, but the GBM was trained on a different set of features. The SHAP values paint a story of the applicant being lower-risk on all considered attributes. For the applicant at the 50th percentile, we see most local feature importance values staying close to their respective intercepts, and for the high-risk applicant nearly all local feature importance values are positive, pushing predictions higher, and both models seem to agree that it's the applicant's recent payment statuses (`PAY_0`, `PAY_2`, and `PAY_3`) that are driving risk, with the GBM and SHAP also focusing on payment amount information.

These are just two types of local feature importance, and we're going to discuss many others just below. But this small example likely brings up some questions that might get stuck in reader's heads if we don't address them before moving on. First and foremost, the act of generating explanations does not mean these are good models, it simply means we get to make a more informed decision about whether the models are good or not. Second, it's not uncommon for two different models to give different explanations for the same row of data. But just because it's common, doesn't make it right. It's not something we should just accept and move past. While these models appear to show adequate agreement when operating on the same features (maybe readers disagree?), the authors argue that, sadly, this is a best case scenario for post-hoc explanation and it happens because we picked relatively simple models and made sure the sign of the GLM coefficients and the direction of the monotonicity con-

straints of the GBM agreed with domain expertise. For complex, black-box models trained on hundreds or more correlated features, we'll likely see much less agreement between model explanations, and that should raise red flags.

There are at least two deeper concepts at play here. One is *consistency*, or the desirable property that different models yield similar explanations for the same row of data. The other concept is known as the *true to data vs. true to model* debate. Most post-hoc explanation techniques discussed below tell us about a specific model's behavior, which can be useful information, and these techniques are called *true to model*. But some techniques, like some forms of SHAP, and logistic regression, are *true to data* and have the ability to tell us about trends and phenomenon in training data. We'll touch on these topics in more detail at the end of the explanation section, but for now a good lesson to learn is that *true to model* explanations are less likely to be consistent because they focus on the, often wild and wrong, local behavior of specific ML models. Consistency seems a reasonable goal for high-risk applications, for many reasons, including bolstering trust in outcomes and agreement between multiple important decision-making systems. If we want consistent explanations use simpler models that are tethered to domain expertise, and use *true to data* explanation techniques that pick up on phenomenon in training data that should be represented in the behavior of multiple models.



True to data explanations attempt to tell us about information in data that many different types of models can learn. *True to model* explanations attempt to summarize the behavior of a specific model.

As readers are probably starting to pick up, post-hoc explanation is complex and fraught, and this is why the authors try to pair these techniques with explainable models so the models can be used to check the explanations and vice-versa. Nevertheless, we push onward to an outline of many major local explanation techniques that either bear a resemblance to local feature importance or implement novel ways to calculate local feature importance: counterfactual, gradient-based, occlusion, prototypes, SHAP and several others.

- **Counterfactuals:** Counterfactual explanations tell us what input features value would have to be changed to change the outcome of a model prediction. The bigger the swing of the prediction when changing an input variable by some standard amount, the more important that feature is as measured from the counterfactual perspective. Check out Christoph Molnar's *Interpretable Machine Learning* section on [counterfactual explanations](#) for more details. Also, note that some researchers claim that counterfactuals are more interpretable to humans than other types of local explanations. However, these claims are not yet uni-

formly accepted. To try out counterfactual explanations, check out [DiCE](#), diverse counterfactual explanations for ML classifiers from Microsoft Research.

- **Gradient-based feature attribution:** Think of gradients as regression coefficients for every little piece of a complex machine-learned function. Gradients can be used on any differentiable function to understand the direction of change at an arbitrary location on a response surface. In deep learning, gradient-based approaches to local explanations are common, and gradients of the trained model with respect to model weights have become commonly available in many deep learning toolkits. When used for image or text data, gradients can often be overlaid on input images and text to create highly-visual explanations depicting where a function exhibits large gradients when creating a prediction for that input. Moreover, various tweaks to gradients are claimed to result in improved explanations, such as [integrated gradients](#), [layerwise relevance propagation](#), [DeepLift](#), and [Grad-CAM](#). For an excellent but highly-technical review of gradient-based explanations, see Ancona et al.'s [*Towards Better Understanding of Gradient-based Attribution Methods for Deep Neural Networks*](#). To see what can go wrong with saliency maps, see [*Sanity Checks for Saliency Maps*](#). We'll return to this important paper many times in subsequent chapters. Also, remember that gradient-based methods are an extremely poor fit for non-differentiable models, like standard decision trees.
- **Occlusion:** Occlusion refers to the simple and powerful idea of removing features from a model prediction and tracking the resulting change in the prediction. A big change may mean the feature is important, a small change may mean it's less important. Occlusion is the basis of SHAP, leave-one-feature-out (LOFO), and many other explanation approaches, including many in computer vision and natural language processing. Occlusion can be used to generate explanations in these complex models when gradients are unavailable for complex model pipelines. Of course, it's never simple mathematically to remove inputs from a model, and it takes a lot of care to generate relevant explanations from the results of feature removal. For an authoritative review of occlusion and feature removal techniques, see Covert, Lundberg, and Lee's [*Explaining by Removing*](#), where they cover 25 explanation methods that can be linked back to occlusion and feature removal.
- **Prototypes:** Prototypes are instances of data that are highly representative of larger amounts of data. Prototypes are used to explain by summarization and comparison. A common kind of prototype is k -means (or other) cluster centroids. These prototypes are an average representation of a similar group of data. They can be compared to other points in their own cluster and in other clusters based on distances and in terms of real-world similarity. Real-world data is often highly heterogeneous and it can be difficult to find prototypes that represent an entire data set well. *Criticisms* are data points that are not represented well by

prototypes. Together, prototypes and criticisms create a set of points that can be leveraged for summarization and comparison purposes to better understand both data sets and ML models. Moreover, several types of ML models, like k -NN and this-looks-like-that deep learning, are based on the notion of prototypes, which enhances their overall interpretability. To learn more about prototypes, look into Molnar's chapter on [prototypes and criticisms](#).

There are various other local explanation techniques. Readers may have heard of [treeinterpreter](#) or [eli5](#), which decompose feature contributions to decision tree ensembles into *true to model*, locally-accurate, additive explanatory values. [Alethia](#) is a similar package for rectified linear unit (ReLU) neural networks.

Next, we'll devote a section to the discussion of Shapley values, one of most popular and rigorous types of local explanations available to data scientists. Before moving on, we'll remind readers once more that these post-hoc explanation techniques, including SHAP, are not magic. While the ability to understand which features influences ML model decisions is a an incredible breakthrough, there is a great deal of contestational literature pointing towards flaws and problems in these techniques. To get the most out of them, approach them with a staid and scientific mindset. Do experiments. Use explainable models and simulated data to assess explanation quality and validity. Does the explanation technique we've selected give compelling explanations on random data? (If so, that's bad.) Does the technique provide stable explanations when data is mildly perturbed? (That's a good thing, usually.) Nothing in life or ML is perfect, and that certainly includes local post-hoc explanation.

Shapley values. Shapley values were created by the Nobel-laureate economist and mathematician Lloyd Shapley. Shapley additive explanations (SHAP) [unify](#) approaches such as LIME, LOFO, [treeinterpreter](#), DeepLift and others to generate accurate local feature importance values and they can be aggregated or visualized to create consistent global explanations. Aside from their own Python package, [shap](#) — and various R packages, SHAP is supported in popular machine learning software frameworks like H2O.ai, LightGBM, and XGBoost.

SHAP starts out the same as many other explanation techniques, asking the intuitive question: what would the model prediction be for this row without this feature? So why is SHAP different from other types of local explanations? To be exact, in a system with as many complex interactions as a typical ML model, that simple question must be answered using an average of all possible sets of inputs that do not include the feature of interest. Those different groups of inputs are called *coalitions*. For a simple data set with, say, twenty columns, that means about half a million different model predictions on different coalitions are considered in the average. Now repeat that process of dropping and averaging for every prediction in our dataset, and we can see why SHAP takes into account much more information than most other local feature importance approaches.

There are many different flavors of SHAP, but the most popular are Kernel SHAP, Deep SHAP and Tree SHAP. Of these, Tree SHAP is exact, and Kernel and Deep SHAP are approximate. Kernel SHAP has the advantage of being usable on any type of model, i.e. being *model-agnostic*. It's like local interpretable model-agnostic explanations (LIME) combined with the coalitional game theory approach. However, with more than a few inputs, Kernel SHAP often requires untenable approximations to achieve tolerable run-times. Kernel SHAP also requires the specification of *background data*, or data that is used by an explanation technique during the process of calculating explanations, which can have a large influence on the final explanation values. Deep SHAP also relies on approximations and may be less suitable than easier-to-compute gradient-based explanations depending on the model and dataset at hand. On the other hand, Tree SHAP is perhaps the most robust post-hoc explanation method available to data scientists today. As the "tree" part of the name indicates, it's only suitable for tree-based models. And the SHAP part of the name should indicate that this approach has a lot going on — it's sophisticated, but not perfect.



Background data is data used to generate explanations, such as the altered data used to generate partial dependence values or the data we select to enable SHAP to integrate features out of a model's conditional distribution. Background data can have a large effect on explanations and must be chosen carefully not conflict with statistical assumptions of explanation techniques and to provide the right context for explanations.

Two of the major places where data scientists tend to go wrong with Tree SHAP are interpretation of SHAP itself and failing to understand the assumptions inherent in different hyperparameter settings. For interpretation, let's start with recognizing SHAP as an offset from the average model prediction. SHAP values are calculated in reference to that offset, and large SHAP values mean the feature causes the model prediction to depart from the average prediction in some noticeable way. Small SHAP values mean the feature doesn't move the model prediction too far from the average prediction. We're also often tempted to read more into SHAP than is actually there. We tend to seek causal or counterfactual logic from SHAP values, and this is simply not possible. SHAP values are the weighted average of the feature's contribution to model predictions across a vast number of coalitions. They don't provide causal or counterfactual explanations, and we'd argue that if we'd even like for them to make sense to other technicians, the underlying model must also make sense.



A SHAP value can be interpreted as the difference in model outcome away from the average prediction attributed to a certain input feature value.

Tree SHAP also asks users to make trade-offs. Based on how features are perturbed, to remove them from a trained model, we are choosing between different philosophies of explanations and different shortcomings. With the default settings in Tree SHAP, `tree_path_dependent` perturbation uses information learned during tree training to perturb and remove features. However, it is less trustworthy in the presence of correlated inputs and, in part because it uses training data information to remove features, `tree_path_dependent` feature perturbation tends to result in explanations that are at least somewhat *true to data*. *True to data* explanations attempt to summarize salient phenomenon in training data as they are represented by the model. (The extent to which Tree SHAP is actually *true to data* is discussed briefly at the end of the chapter and similar issues are discussed in [Chapter 6](#). Put simply, it's not perfect.) The other option for feature perturbation is known as `interventional`. This type of feature perturbation is dependent on humans selecting background data to use when removing features. For explanations to be logical, that background data must provide the correct context to calculate SHAP values, which is a complex mental exercise even for experienced practitioners. `interventional` feature perturbation is also known to be less trustworthy when sampling from background data leads to unrealistic combinations of feature values in SHAP calculations, but it does help alleviate problems that arise from correlation between input features and is seen as being more specifically related to the model at hand, i.e., more *true to model*.



`tree_path_dependent` SHAP values are thought to be less robust to correlation. `interventional` SHAP values are subject to issues relating to unrealistic combinations of values sampled from background data.



Due to general issues with correlation, information overload, and commonsense, good explanations usually require that the underlying model is trained on a smaller number of uncorrelated features with a direct relationship to the modeling target.

This web of assumptions and limitations mean that we still have to be careful and thoughtful, even when using Tree SHAP. We can make things easier on ourselves though. Correlation is the enemy of many explainable models and post-hoc explanation techniques, and SHAP is no different. The authors like to start with a reasonable number of input features that do not have serious multicollinearity issues. On a good day, we'd have found those features using a causal discovery approach as well. Then we'd use domain knowledge to apply monotonicity constraints to input features in XGBoost. For general feature importance purposes, we'd use Tree SHAP with `tree_path_dependent` feature perturbation. This way our explanations should make sense, because they're linked to logical, observable real-world phenomena in data,

and not contaminated by too much correlation. For an application like credit scoring, where the proper context is defined by regulatory commentary, we might use *interventional* SHAP values and background data. For instance, regulatory commentary on the generation of explanations for credit denials in the U.S. *suggests*, “to identify the factors for which the applicant’s score fell furthest below the average score for each of those factors achieved by applicants whose total score was at or slightly above the minimum passing score.” This means our background data set should be composed of applicants with predictions just above the cutoff to receive the credit product.

Critical Applications of Local Explanations and Feature Importance. Meeting regulatory requirements for algorithmic transparency is likely the most mission-critical application of local feature importance values. The primary requirement now in the U.S. is to explain credit denials with *adverse action notices*. The key technical component for the adverse action reporting process are *reason codes*. Reason codes are plain text explanations of a model prediction described in terms of a model’s input features. They are a step beyond local feature importance, in which raw local feature importance values are matched to reasons a product can be denied. Consumers should then be allowed to review the reason codes for their negative prediction and follow a prescribed appeal process if data inputs or decision factors are demonstrably wrong.

Adverse action reporting is a specific instance of a more high-level notion known as *actionable recourse*, where transparent model decisions are based on factors users have control over and can be appealed by model users and overridden by model operators. Many forthcoming and proposed regulations, such as emerging regulations in **California, Washington, DC** and the **E.U.** are likely to introduce similar requirements for explanation or recourse. When working under regulatory scrutiny, or just to do the right thing when making important decisions for other human beings, we’ll want our explanations to be as accurate, consistent, and interpretable as possible. While we expect that local feature importance will be one of the most convenient technical tools to generate the raw data needed to comply, we’ll make the best explanations when combining local feature importance with explainable models and other types of explanations, like those described in the subsequent sections of this chapter.

Global Feature Importance

Global feature importance methods quantify the global contribution of each input feature to the predictions of a complex ML model over an entire dataset, not just for one individual or row of data. Global feature importance measures sometimes give insight into the average direction that a variable pushes a trained ML function, and sometimes they don’t. At their most basic, they simply state the magnitude of a feature’s relationship with the response as compared to other input features. This is hardly ever a bad thing to know, and since most global feature importance measures are

older approaches, they are often expected by model validation teams. Figure [Figure 2-5](#) below provides an example of feature importance plots, in which we are comparing the global feature importance of two models.

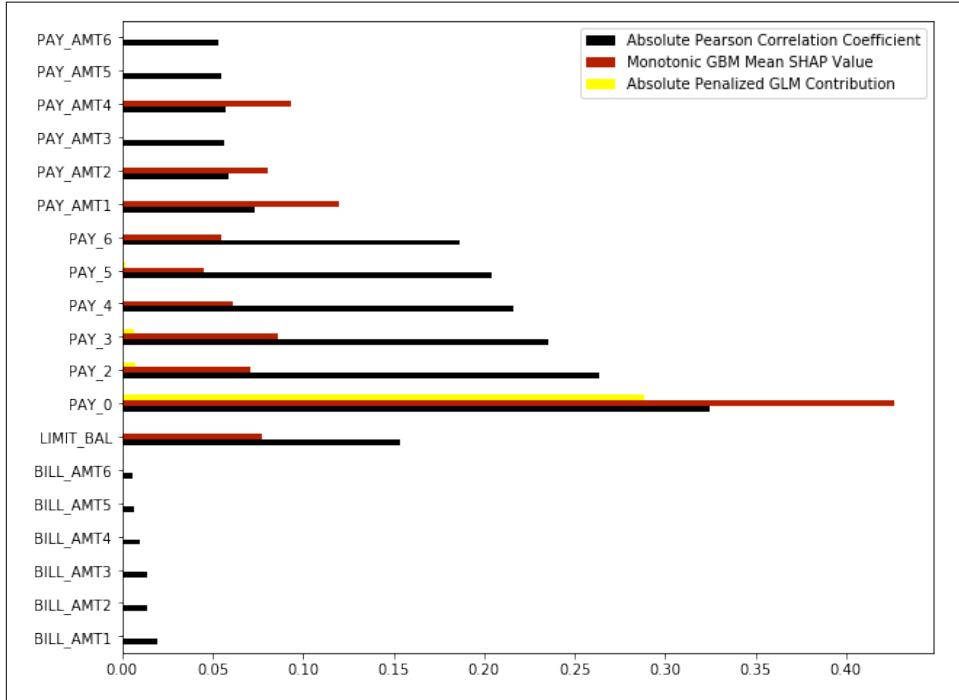


Figure 2-5. Global feature importance for two explainable models compared to Pearson correlation.

Charts like this help us answer questions like:

- Does one ordering of feature importance make more sense than the other?
- Does this plot reflect patterns we know the models should have learned from training data?
- Are the models placing too much emphasis on just one or two features?

Global feature importance is a straightforward way to conduct such basic sanity checks. In Figure [Figure 2-5](#), we compare feature importance to Pearson correlation to have some baseline understanding of which features should be important. And between the Pearson correlation and the two models, we can see everyone agrees that PAY_0 is the most important feature. However, the GLM places nearly all of its decision-making on PAY_0, while the GBM spreads importance over a larger set of inputs. When models place too much emphasis on one feature, as the GLM in Figure

Figure 2-5 does, it can make them unstable in new data if the distribution of the most important feature drifts, and it makes adversarial manipulation of a model easy. For the GLM in Figure **Figure 2-5**, a bad actor would only have to alter the value of a single feature, PAY_0, to drastically change the model's predictions.

Global feature importance metrics can be calculated in many ways. Many data scientists are first introduced to feature importance when learning about decision trees. A common feature importance method for decision trees is to sum up the change in the splitting criterion for every split in the tree based on a certain feature. For instance, if a decision tree (or tree ensemble) is trained to maximize information gain for each split, the feature importance assigned to some input feature is the total information gain associated with that feature every time it is used in the tree(s). Perturbation-based feature importance is another common type of feature importance measurement. While perturbation-based feature importance is often introduced alongside split-based feature importance for decision trees, it's actually a *model-agnostic* technique, meaning it can be used for almost all types of ML models. In perturbation-based feature importance, an input feature of interest is shuffled (sorted randomly) and predictions are made. The difference in some original score, usually the model prediction or something like mean squared error (MSE), before and after shuffling the feature of interest is the feature importance. Another similar approach is known as leave-one-feature-out (LOFO, or leave-one-covariate-out, LOCO). In the LOFO method, a feature is somehow dropped from the training or prediction of a model — say by retraining without the feature and making predictions, or by setting the feature to missing and making predictions. The difference in some score between the model with the feature of interest and without the feature of interest is taken to be the LOFO importance.

While permutation and LOFO are typically used to measure the difference in predictions or the difference in some accuracy or error score, they have the advantage of being able to estimate the impact of a feature on nearly anything associated with a model. For instance, it's quite possible calculate permutation or LOFO based contributions to a fairness metric. That allows us to gain insight into which specific features are contributing to any detected sociological bias. Using permutation as the example approach, consider calculating a fairness metric like adverse impact ratio (AIR) for women vs. men with the original features and then taking the second step to calculate female vs. male AIR based on those predictions. If we wanted to see how much an input feature, like credit score, is contributing to female vs. male AIR, one approach would be to randomly shuffle credit score, regenerate predictions based on the permuted credit score feature, recalculate female vs. male AIR, and then examine the difference between the original AIR score and the new AIR score. If permuting a feature causes a large change in the AIR value, it's probably one of the features driving issues with bias. This same motif can be re-applied for any number of measures of interest about a model — error functions, security, privacy and more.



Techniques like perturbation feature importance and LOFO can be used to estimate contributions to many quantities besides model predictions.

Because these techniques are well-established, we can find a great deal of related information and software packages. For a great discussion on split-based feature importance, check out [Chapter 3](#) of an *Introduction to Data Mining*. Section 10.13.1 of *Elements of Statistical Learning* introduces split-based feature importance, and Section 15.3.2 provides a brief introduction to permutation-based feature importance in the context of random forests. The R package [vip](#) provides a slew of *variable importance plots*, and we can try LOFO with the Python package [lofo-importance](#). Of course, there are drawbacks and weaknesses to most global feature importance techniques. Split-based feature importance has serious consistency problems, and like so much of post-hoc XAI, correlation will lead us astray with permutation-based and LOFO approaches. The paper [There Is No Free Variable Importance](#) gets into more details of the sometimes disqualifying issues related to global feature importance. But, as repeated many times in this chapter, constrained models with a reasonable number of non-correlated and logical inputs will help us avoid the worst problems with global feature importance.

SHAP also has a role to play in global feature importance. SHAP is by nature a local feature importance method, but it can be aggregated and visualized to create global feature importance information. Of the many benefits SHAP presents over more traditional feature importance measures, its interpretation is perhaps the most important. With split-based, permutation, and LOFO feature importance, often times we only see a relative ordering of the importance of the input features, and maybe some qualitative notions of how a feature actually contributes to model predictions. With SHAP values, when calculating their mean absolute value per feature, this quantity is the mean absolute contribution of a feature to model predictions — a quantity with a clear and quantitative relationship to model predictions. SHAP also provides many levels of granularity for feature importance. While SHAP can be directly aggregated into a feature importance value, that process can average out important local information. SHAP opens up the option of examining feature importance values anywhere from the most local level — a single row — to the global level. For instance, aggregating SHAP across important segments, like U.S. states or different genders, or using the numerous visualizations in the `shap` package, can provide a medium-level view of feature importance that might be more informative and more representative than a single mean absolute contribution to prediction value. Like permutation and LOFO feature importance, SHAP can also be used to estimate importance to quantities besides model predictions. It's capable of estimating contributions to [model errors](#) and to fairness metrics, like [demographic parity](#).

This concludes our discussion of feature importance. And whether it's global or local, feature importance will probably be the first post-hoc XAI technique we encounter when we're building models. As this section shows, there's a lot more to feature importance than just bar charts or running SHAP. To get the best results with feature importance, we'll have to be familiar with the strengths and weaknesses of the many approaches. Next, we'll be covering surrogate models — another intriguing explanation approach, but also one that requires thought and caution.

Surrogate Models

Surrogate models are simple models of complex models. If we can build a simple, interpretable model of a more complex model, we can use the explainable characteristics of the surrogate model to explain, summarize, describe or debug the more complex model. Surrogate models are generally *model agnostic*. We can use them for almost any ML model. The problem with surrogate models is they are mostly a trick-of-the-trade technique, with few mathematical guarantees that they truly represent the more complex model they are attempting to summarize. That means we have to be careful when using surrogate models, and at a minimum, check that they are accurate and stable representations of the more complex models they seek to summarize. In practice, this often means looking at different types of accuracy and error measures on many different partitions and folds to ensure fidelity to the more complex model's predictions is high, and that it remains high in new data and stable during cross-validation. Surrogate models also have many names. Readers may have heard about *model compression*, *model distillation*, or *model extraction*. All of these either are surrogate modeling techniques or are closely related. Like feature importance, there are also many different types of surrogate models, and they can be thought of in terms of global or local as well. In the sections below, we'll start out with decision tree surrogate models, which are typically used to construct global explanations, then transition into local interpretable model-agnostic explanations (LIME) and anchors, which are typically used to generate local explanations.

Decision Tree Surrogates

Decision tree surrogate models are usually created by training a decision tree on the original inputs and predictions of a complex model. Feature importance, trends, and interactions displayed in the surrogate model are then assumed to be indicative of the internal mechanisms of the complex model. There are few, and possibly no, theoretical guarantees that the simple surrogate model is highly representative of the more complex model. But, because of the structure of decision trees, these surrogate models create very interpretable flow charts of a more complex model's decision-making processes, as is visible in Figure [Figure 2-6](#) below. There are prescribed methods for training decision tree surrogate models, for example those explored in [Extracting](#)

Tree-Structured Representations of Trained Networks and Interpretability via Model Extraction.



Decision tree surrogate models can be highly interpretable when they create flow charts of more complex models.

In practice, it usually suffices to measure the fidelity of the surrogate tree's predictions to the complex model's predictions in the data partition of interest with metrics like logloss, root mean square error (RMSE) or R^2 and to measure the stability of those predictions with cross-validation. If a surrogate decision tree fails to provide high-fidelity with respect to the more complex model, more sophisticated explainable models, like EBM or XNN, can be considered as surrogates instead.

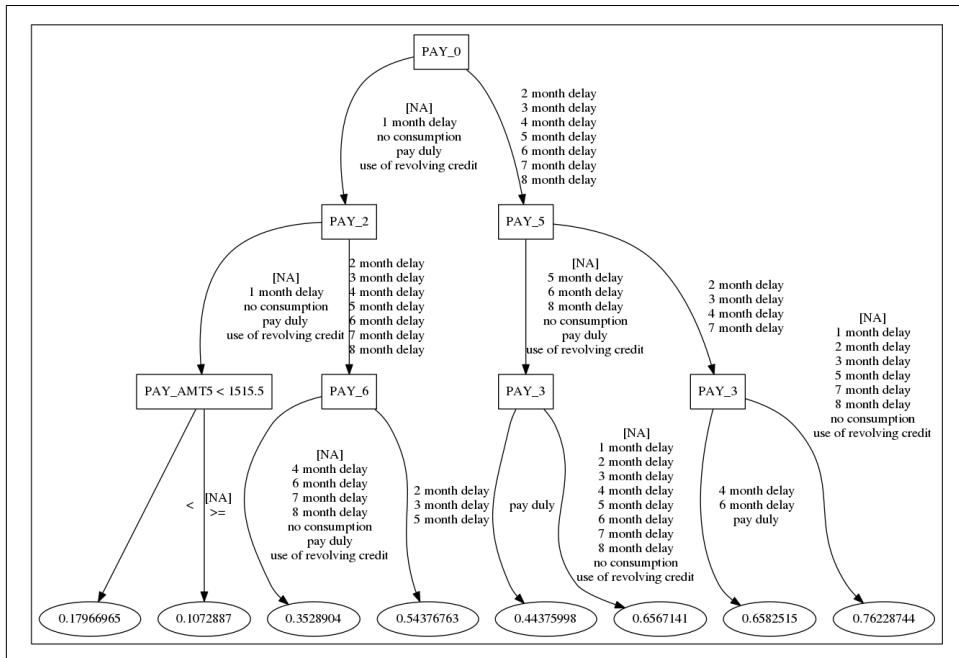


Figure 2-6. A decision tree surrogate model creates a flowchart for a monotonic gradient boosting machine (GBM).

The surrogate model in Figure 2-6 was trained on the same inputs as the more complex GBM it seeks to summarize, but instead of training on the original target that indicates payment delinquency, it is trained instead on the predictions of the GBM. When interpreting this tree, features that are higher or used more frequently are considered to be more important in the explained GBM. Features that are above

and below one another can have strong interactions in the GBM, and other techniques discussed in this chapter such as explainable boosting machines and a comparison between partial dependence and ICE can be used to confirm the existence of the interaction in training data or in the GBM model.



Decision tree surrogates can be used to find interactions for use in linear models on for LIMEs. EBMs and differences between partial dependence and ICE can also be used to find interactions in data and models.

The decision paths in the tree can also be used to gain some understanding of how the more complex GBM makes decisions. Tracing the decision path from the root node in Figure [Figure 2-6](#) to the average predictions at the bottom of the tree, we can see that those who have good statuses for their most recent (`PAY_0`) and second (`PAY_2`) most recent repayments and have somewhat large fifth most recent payment amounts (`PAY_AMT5`) are most likely not to have future delinquency. Those customers who have unfavorable most and fifth most recent repayment statuses appear most likely to have future payment problems. (The `PAY_3` splits exhibit a large amount of noise and are not interpreted here.) In both cases, the GBM appears to be considering both recent and past repayment behaviors to come to its decision about future payments. This prediction behavior is logical but should be confirmed by other means when possible. Like most surrogate models, decision tree surrogates are useful and highly interpretable, but should not be used for important explanation or interpretation tasks on their own.

Linear Models and Local Interpretable Model-agnostic Explanations

LIME seems to be the first, the most famous, and the most criticized post-hoc explanation technique to be released in recent years. As the name indicates, it's most often used for generating local explanations, by fitting a linear model to the predictions of some small region of a more complex model's predictions. While this is its most common usage, it's a reductionist take on the technique. When first introduced in the 2016 article [*Why Should I Trust You?: Explaining the Predictions of Any Classifier*](#), LIME was presented as a framework with several admirable qualities. The most appealing of which was a sparsity requirement for local explanations. If our model has 1000 features and we apply SHAP, we will get back 1000 SHAP values for every prediction we want to explain. Even if SHAP is perfect for our data and model, we'll still have to sort through 1000 values every time we want to explain some prediction. The framework of LIME circumvents this problem by requiring that generated explanations are sparse, meaning that they key into the small handful of locally-important features instead of all the features included in the model to be explained. The rest of the framework of LIME specifies fitting an interpretable surrogate model

to some weighted local region of another model's predictions. And that's a more faithful description of the LIME framework — a locally-weighted interpretable surrogate model with a penalty to induce sparsity, fit to some arbitrary more complex model's predictions. These ideas are useful and quite reasonable.



A LIME value can be interpreted as the difference between a LIME prediction away from the associated LIME intercept attributed to a certain input feature value.



Always ensure LIME fits the underlying response function well with fit statistics and visualizations, and that the local model intercept is not explaining the most salient phenomenon driving a given prediction.

It's the popular implementation of LIME that gets inexperienced users into trouble and that presents security problems. For tabular data, the software package `lime` asks users to select a row to be explained, generates a fairly simplistic sample of data based on a specified input dataset, weights the sample by the user-selected row, fits a least absolute shrinkage and selection operator (LASSO) regression between the weighted sample and the more complex model's predictions on the sample, and finally, uses the LASSO regression coefficients to generate explanations for the user-specified row. There's a lot of potential issues in that implementation:

- The sampling is a problem for real-time explanation because it requires data generation and fitting a model in the midst of a scoring pipeline, and it also opens users up to data poisoning attacks that can alter explanations.
- Generated LIME samples can contain large proportions of out-of-range data that can lead to unrealistic local feature importance values.
- Local feature importance values are offsets from the local GLM intercept, and this intercept can sometimes account for the most important local phenomena.
- Extreme non-linearity and high-degree interactions in the selected local region of predictions can cause LIME to fail completely.

Because LIME can be used on almost any type of ML model to generate sparse explanations, it can still be a good tool in our kit if we're willing to be patient and think through the LIME process. If we need to use LIME, we should plot the LIME predictions versus our more complex model predictions and analyze them with RMSE, R² or similar. We should be careful about the LIME intercept and make sure it's not explaining our prediction on its own, rendering the actual LIME values useless. To increase the fidelity of LIME, try LIME on discretized input features and on

manually constructed interactions. (We can use decision tree surrogates to guess at those interactions.) Use cross-validation to estimate standard deviations or even confidence intervals for local feature contribution values. And keep in mind that poor fit or inaccuracy of local linear models is itself informative, often indicating extreme nonlinearity or high-degree interactions in that region of predictions.

Anchors and Rules

On the heels of LIME, probably with some lessons in mind, the same group of researchers released another model-agnostic local post-hoc explanation technique named anchors. Anchors generates high-fidelity sets of plain-language rules to describe a machine learning model prediction, with a special focus on finding the most important features for the prediction at hand. Readers can learn more about Anchors in *Anchors: High-Precision Model-Agnostic Explanations* and the software package `anchor`. While anchors is a prescribed technique with documented strengths and weaknesses, it's just one special instance of using rule-based models as surrogate models. As discussed in the first part of the chapter, rule-based models have good learning capacity for nonlinearities and interactions, while still being generally interpretable. Likely many of the rule-based models highlighted previously could be evaluated as surrogate models.

Plots of Model Performance

In addition to feature importance and surrogate models, partial dependence, individual conditional expectation (ICE) and accumulated local effect (ALE) plots have become popular for describing trained model behaviors with respect to input features. In this section, we'll go over partial dependence and ICE, how partial dependence should really only be used with ICE, and discuss ALE as a more contemporary replacement for partial dependence.

Partial Dependence and Individual Conditional Expectation

Partial dependence plots show us the estimated average manner in which machine-learned response functions change based on the values of one or two input features of interest, while averaging out the effects of all other input features. Remember the averaging out part. We'll circle back to that. Partial dependence plots can show the nonlinearity, nonmonotonicity, and two way interactions in complex ML models and can be used to verify monotonicity of response functions trained under monotonicity constraints. Partial dependence is introduced along with tree ensembles in *Elements of Statistical Learning*, Section 10.13. ICE plots are a newer, local, and less well-known adaptation of partial dependence plots. They depict how a model behaves for a single row of data. ICE pairs nicely with partial dependence in the same plot to provide more local information to augment the more global information provided by partial dependence. ICE plots were introduced in the paper *Peeking Inside the Black Box: Vis-*

ualizing Statistical Learning with Plots of Individual Conditional Expectation. There are lots of software packages for us to try partial dependence and ICE. For Python users, check out [PDPbox](#) and [PyCEbox](#). For R users, there are the [pdp](#) and [ICEbox](#) packages. Also many modeling libraries support partial dependence without having to use an external package.

Partial dependence should be paired with ICE plots, as ICE plots can reveal inaccuracies in partial dependence due to the averaging out of strong interactions or the presence of correlation. When ICE curves diverge from partial dependence curves, this can indicate strong interactions between input features, which is another advantage of using them together. We can then use EBM or surrogate decision trees to confirm the existence of the interaction in training data or the model being explained. One more trick is to plot partial dependence and ICE with a histogram of the feature of interest. That gives good insight into whether any plotted prediction is trustworthy and supported by training data. In Figure [Figure 2-7](#) below, partial dependence, ICE and a histogram of PAY_0 are used to summarize the behavior of a monotonic GBM.

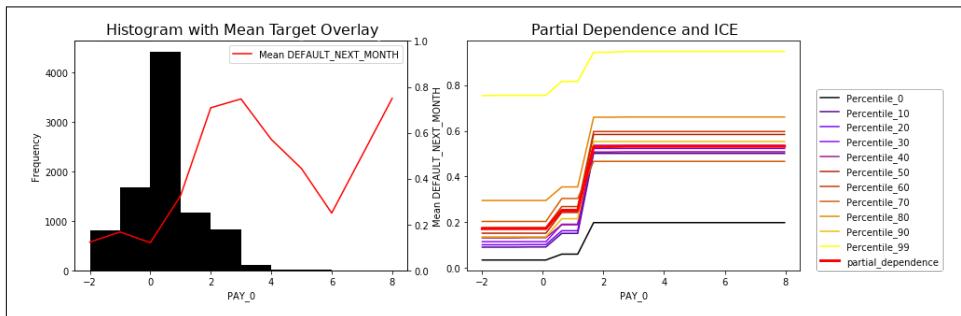


Figure 2-7. Partial dependence and ICE for an important input variable with accompanying histogram and mean target value overlay for a monotonic gradient boosting machine (GBM).

On the left, we can see a histogram of PAY_0 and that there is simply not much data for customers who are more than two months late on their most recent payment. On the right we see partial dependence in red, and ICE curves for customers at the deciles of predicted probability of default. Partial dependence and ICE help us confirm the monotonicity of the constrained GBM's response function for PAY_0. The model appears to behave reasonably, even when there is not much data to learn from for higher values of PAY_0. Probability of default increases in a monotonic fashion as customers become more late on their most recent payment, and probability of default is stable for higher values of PAY_0, even though there is almost no data to support the classifier in that region. It might be tempting to believe that every time we use monotonic constraints we would be protected against our ML models learning silly behaviors when there isn't much training data for them to learn from, but this is not true.

Yes, monotonic constraints help with stability and underspecification problems, and partial dependence and ICE help us spot these problems if they occur, but we got lucky here. The truth is we need to check all our models for unstable behavior in sparse domains of the training data, and be prepared to have specialized models, or even human case workers, ready to make good predictions for these difficult rows of data.



Comparing explainable model shape functions, partial dependence, ICE, or ALE plots with a histogram can give a basic qualitative measure of uncertainty in model outcomes, by enabling visual discovery of predictions that are based on only small amounts of training data.



Due to multiple known weaknesses, partial dependence should not be used without ICE, or ALE should be used in place of partial dependence.

Here's one more word of advice before moving onto ALE plots. Like feature importance, SHAP, LIME, and all other explanation techniques that operate on a background dataset, we have to think through issues of context with partial dependence and ICE. They both use sneaky implicit background data. For partial dependence, it's whatever data set our interested in with all the values of the feature being plotted set to a certain value. This alters patterns of interactions and correlation, and although an exotic concern, opens us up to data poisoning attacks as addressed in *Fooling Partial Dependence via Data Poisoning*. For ICE, the implicit background data set is a single row of data with the feature of interest set to a certain value. Watch out for the ICE values being plotted being too unrealistic in combination with the rest of the observed data in that row.

Accumulated Local Effect

ALE is a newer and highly rigorous method for representing the behavior of an ML model across the values of an input feature, introduced in *Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models*. Like partial dependence plots, ALE plots show the shape — i.e., nonlinearity or nonmonotonicity — of the relationship between predictions and input feature values. ALE plots are especially valuable when strong correlations exist in the training data, a situation where partial dependence is known to fail. ALE is also faster to calculate than partial dependence. Try it with [ALEPlot](#) in R, and the Python edition [ALEPython](#).

Cluster profiling

While a great deal of focus has been placed on explaining supervised learning models, sometimes we need to use unsupervised techniques. Feature extraction and clustering are two of the most common unsupervised learning tasks. We discussed how to make feature extraction more explainable with sparse methods like SPCA and NMF when we covered explainable models. And with the application of very much established post-hoc methods of profiling, clustering can often be made more transparent too. The simplest approach is to use means and medians to describe cluster centroids, or to create prototypical members of a dataset based on clusters. From there, we can use concepts associated with prototypes such as summarization, comparison, and criticisms to better understand our clustering solution. Another technique is to apply feature extraction, particularly sparse methods, to project a higher dimensional clustering solution into two or three dimensions for plotting. Once plotted on sparse interpretable axes, it's easier to use our domain knowledge to understand and check a group of clusters. Distributions of features can also be employed to understand and describe clusters. The density of a feature within a cluster can be compared to its density in other clusters or to its overall distribution. Features with the most dissimilar distributions versus other clusters or the entire training data can be seen as more important to the clustering solution. Finally, surrogate models can be applied to explain clusters. Using the same inputs as the clustering algorithm and the cluster labels as the target, fit an interpretable classifier like a decision tree to our clusters and use the surrogate model's interpretable characteristics to gain insight into our clustering solution.

Stubborn Difficulties of Post-hoc Explanation in Practice

Unless we're careful, we can get into very murky areas with post-hoc explanation. We've taken care to discuss the technical drawbacks of the techniques, but there is even more to consider when working with these techniques on real-world high-risk applications. As a refresher, Professor Rudin's *Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead* lays out the primary criticisms of post-hoc explanation for black-box ML models and high risk uses. According to Rudin, explanations for *black-box* models are:

- Premised on the wrong belief that black-boxes are more accurate than explainable models.
- Not faithful enough to the actual innerworkings of complex black-box models.
- Often nonsensical.
- Difficult to calibrate against external data.
- Unnecessarily complicated.

To be clear, this chapter has not advocated for the use of post-hoc explanation with black-box models, but instead with explainable models, where the model and the explanations can act as process controls for one another. Even using explanations in this more risk-aware manner, there are still serious issues to address. This section will highlight the concerns we see the most in practice. We'll close this section by highlighting the advantages of using explainable models and post-hoc explanations in combination. But as the transparency case will show, even if we get things mostly right on the technical side of transparency, human factors are still immensely important to the final success, or failure, of a high-stakes ML application.

Christoph Molnar has not only been prolific in teaching us how to use explanations, he and co-authors have also been quite busy researching their drawbacks. If readers would like to dig into details of issues with common explanation approaches, we'd suggest both *General Pitfalls of Model-Agnostic Interpretation Methods for Machine Learning Models* and the earlier *Limitations of Interpretable Machine Learning Methods*. Below we'll outline the issues we see the most in our practice: confirmation bias, context, correlation and local dependence, hacks, human interpretation, inconsistency, and measuring the fidelity of explanations

- **Confirmation Bias:** For most of this chapter, we've discussed increased transparency as a good thing. While it certainly is, increased human understanding of ML models, and the ability to intervene in the functioning of those models, does open cracks for confirmation bias to sneak into our ML workflow. For example, let's say we're convinced a certain interaction should be represented in a model based on past experience in similar projects. However, that interaction just isn't appearing in our explainable model or post-hoc explanation results. It's extremely hard to know if our training data is biased, and missing the known important interaction, or if we're biased. If we intervene in the mechanisms of our model to somehow inject this interaction, we could simply be succumbing to our own confirmation bias and ruin the model.

Of course, a total lack of transparency also allows confirmation bias to run wild, as we can spin a model's behavior in whatever way we like. The only real way to avoid confirmation bias is to stick to the scientific method, and battle-tested scientific principles like transparency, verification and reproducibility.

- **Context:** *Do Not Explain Without Context* says Dr. Przemyslaw Biecek and team. In practice, this means using logical and realistic background data to generate explanations, and making sure background data cannot be manipulated by adversaries. Even with solid background data for explanations, we still need to ensure our underlying ML models are operating in a logical context too. For us, this means a reasonable number of uncorrelated input features, all with direct relationships to the modeling target.

- **Correlation and dependencies:** While correlation may not prevent ML algorithms from training and generating accurate *in-silico* predictions in many cases, it does make explanation and interpretation very difficult. In large datasets, there are typically many correlated features. Correlation violates the principle of independence, meaning we can't realistically interpret features on their own. When we attempt to remove a feature, as many explanations techniques do, another correlated feature will swoop in and take its place in the model, nullifying the effect of the attempted removal and the removal's intended meaning as an explanation tool. We also rely on perturbing features in explanation, but if features are correlated, it makes very little sense to perturb just one of them to derive an explanation. Worse, when dealing with ML models, they can learn local dependencies, meaning different correlation-like relationships on a row-by-row basis. It's almost impossible to think through the complexities of how correlation corrupts explanations, much less how complex local dependencies might do the same.
- **Hacks:** Explanation techniques that use background data, and many do, can be altered by adversaries. These include LIME and SHAP, as explored in *Fooling LIME and SHAP: Adversarial Attacks on Post hoc Explanation Methods*, and partial dependence, as described in *Fooling Partial Dependence via Data Poisoning*. While these hacks are likely an exotic concern for now, we don't want to be part of the first major hack on ML explanations. Make sure that the code used to generate background data is kept secure, that background data cannot be unduly manipulated during explanation calculations, and that everyone on our team is trusted. Data poisoning, whether against training data or background data, is easy for inside attackers. Even if background data is safe, explanations can still be misinterpreted in malicious ways. In what's known as *fairwashing*, explanations for a sociologically biased ML model are made to look fair, abusing explanations to launder bias while still exposing model users to real harm.
- **Human Interpretation:** ML is difficult to understand, sometimes even for experienced practitioners and researchers. Yet, the audience for ML explanations is much broader than just experienced ML practitioners and researchers. High-risk applications of ML often involve important decisions for other human beings. Even if those other human beings are highly educated attorneys or physicians, we cannot expect them to understand a partial dependence and ICE plot or an array of SHAP values. To get transparency right for high-stakes situations, we'll need to work with psychologists, domain experts, designers, user interaction experts and maybe other types of experts. It will take extra time and product iterations, with tedious communications between technicians, domain experts, and users. Not doing this extra work can result in abject failure, even if technical transparency goals are met, as the case below discusses.

- **Inconsistency:** Consistency refers to stable explanations across different models or data samples. Consistency is difficult to achieve and very important for high-stakes ML applications. In situations like credit or pretrial release decisions, people might be subject to multiple automated decisions and associated explanations, and especially in a more automated future. If the explanations give different reasons for the same outcome decision, this will complicate already difficult situations. To increase consistency, explanations need to key into real, generalizable phenomena in training data and in the application domain. To achieve consistency between our own models, we'll repeat our suggestions related to reasonable numbers of independent, parsimonious input features, but add in model constraints and true to data explanations. Parsimonious models, that are constrained to be more stable and obey reality, paired with more true to data explanations seem more likely to generate consistent explanations even if mechanisms across several models are somewhat different. Conversely, for complex, unconstrained black-box models with numerous and correlated inputs, specters like over-fitting, underspecification and the Rashomon effect make consistent explanations nearly impossible.
- **Measuring Explanation Quality:** Imagine training a model, eyeballing the results, and then assuming it's working properly and deploying it. That's likely a bad idea. But that's how we all work with post-hoc explanations. Given all the technical concerns raised in previous sections, we obviously need to test explanations and see how they work for a given data source and application, just like we do with any other ML technique. Like our efforts to measure model quality before we deploy, we should be making efforts to measure explanation quality too. There are published proposals for such measurements and commonsense testing techniques we can apply. *Towards Robust Interpretability with Self-Explaining Neural Networks* puts forward explicitness, i.e., whether explanations are immediately understandable, faithfulness, i.e., whether explanations are true to known important factors, and stability, i.e., whether explanations are consistent with neighboring data points. *On the (In)Fidelity and Sensitivity of Explanations* introduces related eponymous tests. Beyond these proposals for formal measurement, we can check that explainable model mechanisms and post-hoc explanations confirm one-another, if we use both explainable models and explanations. If older trustworthy explanations are available, we can use those as benchmarks against which to test new explanations for fidelity. Also, stability tests, in which data or model are perturbed in small ways, should generally not lead to major changes in post-hoc explanations.



Test explanations before deploying them in high-risk use cases. While the lack of ground truth for explanations is a difficult barrier, explanations should be compared to interpretable model mechanisms. Comparisons to benchmark explanations, explicitness and fidelity measures, perturbation, comparison to nearest neighbors, and simulated data can also be used to test explanation quality.

There's no denying the hypey appeal of explaining any model, no matter how complex, simply by applying some post-processing. But given all the technical and worldly problems we've just been over, hopefully we've convinced readers that black-box model explanation is really a pipe dream. Explaining black-boxes might not be impossible, but it's technically difficult today, and once we consider all the human factors required to achieve real-world transparency, it becomes even more difficult.

Pairing Explainable Models and Post-hoc Explanation

As we end the chapter's more technical discussions, we'd like to highlight new research that helps elucidates why explaining black-boxes is so difficult and leave us with an example of combining explainable models and post-hoc explanations. Two recent papers relate the inherent complexity of black-box ML models to transparency difficulties. First, *Assessing the Local Interpretability of Machine Learning Models* proxies complexity with the number of runtime operations associated with an ML model decision, and shows that as the number of operations increases, interpretability decreases. Second, *Quantifying Model Complexity via Functional Decomposition for Better Post-hoc Interpretability* uses number of features, interaction strength, and main effect complexity to measure the overall complexity of ML models, and shows that models that minimize these criteria are more reliably interpretable. In summary, complex models are hard to explain and simpler models are easier to explain. Figure Figure 2-8 below provides an example of augmenting a simple model with explanations.

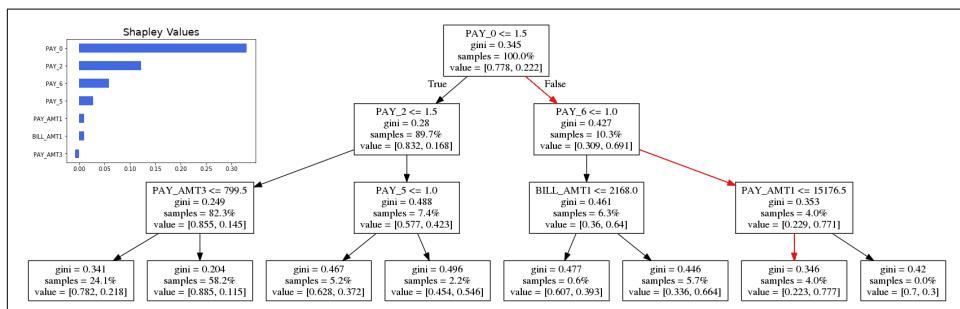


Figure 2-8. A simple explainable model paired with post-hoc explanation information.

Figure [Figure 2-8](#) contains a trained three-level decision tree with a decision path highlighted in red, and the Tree SHAP values for a single row of data that follows that decision path. While Figure [Figure 2-8](#) looks simple, it's actually illustrative of several fundamental problems in ML. Before we dive into the difficulties presented by Figure [Figure 2-8](#), let's bask in the glory of a predictive model with its entire global decision-making mechanism on view, and for which we can generate accurate numeric contributions of the input features to any model prediction. That level of transparency used to be reserved for linear models, but all the new approaches we've covered in this chapter make this level of transparency a reality for a much broader class of higher-capacity models. This means we can train more sophisticated models, that learn more from data, and still be able to interpret and learn from the results ourselves. We can learn more from data, do so reliably, and learn more as humans from the results. That's a huge breakthrough.



Use explainable models and post-hoc explanations together to check one another and to maximize the transparency of ML models.

Now let's dig into the difficulties in Figure [Figure 2-8](#) while keeping in mind these problems always exist when using ML models and post-hoc XAI. (We can just see them and think through them in this simple case.) Notice that the decision path for the selected individual considers PAY_0, PAY_6, and PAY_AMT1. Now look at the Tree SHAP values. They give higher weight to PAY_2 than PAY_6, and weigh PAY_5 over PAY_AMT1, but PAY_2 and PAY_5 are not on the decision path. How can this be? There are two likely causes and they both point to fundamental limitations of ML.

One reason PAY_2 and PAY_5 are included in the Tree SHAP values is we used the more true to data `tree_path_dependent` feature perturbation to create them. This means thousands of versions of this decision tree model were considered to generate these explanations, and this decision tree — learned from our training data — is used to create the required background data for SHAP. In this cloud of models, PAY_2 and PAY_5 were often important factors in the training data for applicants like the applicant under consideration in Figure [Figure 2-8](#). Even if PAY_2 and PAY_5 don't factor heavily into the specific encoding of the training data pictured in Figure [Figure 2-8](#), across many feature coalitions and data samples, PAY_2 and PAY_5 are important factors.



SHAP can be thought of as explaining a family of related models, or as a weighted average of *true to model* explanations across many related models with slightly different input features.

Readers may be thinking, “but I just want to explain *my* model,” and we can do that with something like `treeinterpreter` or `alethia`. But if we do, we run the risk of ignoring the Rashomon effect and underspecification in our explanations. These two intrinsic challenges in ML tell us that even for a laughably simple model and data set, as in Figure [Figure 2-8](#), there are an impossible number of models to choose from. Moreover, unless we use domain constraints and in-domain testing, we won’t pick the right model for deployment. Tree SHAP can actually help account for the near-impossibility of selecting the best model by considering so many different data and model perturbations when generating explanations. If we fall prey to the Rashomon effect and underspecification, which is easy to do, Tree SHAP still provides some hope of having solid explanations. If we were to choose a more true to model explanation technique, then our explanations would likely be more representative the particular encoding of the training data we think is the best, i.e., our “best” model selected via validation data. But if we’re wrong about model selection, then we’re going to have selected an over-fit and underspecified model with the low-quality explanations that are tightly coupled to it.



Much like ensemble models create more stable and accurate predictions, SHAP may create more stable and accurate explanations even if we’ve made some mistakes in specifying our model or selecting the optimal input feature set.

The other reason we see contributions of features not on the decision path in Tree SHAP is because explaining ML models is very hard, and for many different reasons. Despite all the code, math, review, commentary, feedback and iterations that have gone into Tree SHAP and the `shap` package, density estimation in high dimensions is hard — really hard. Tree SHAP must estimate densities to create the background data needed to “remove” a feature to understand its contribution to a single coalition. This crucial step of Tree SHAP relies on the little decision tree in Figure [Figure 2-8](#) to estimate the multidimensional density of background data. How can we hope that the tree structure of our model (trained to make predictions, not estimate conditional densities) encodes this high-dimensional joint distribution well? It likely doesn’t and the quality of the generated SHAP values will suffer. There’s so much to think about on just the technical side of explanations. The case below will dig into human factors of explanations, which might be even more difficult to get right.

Case Study: Graded by Algorithm

Transparency into ML models is no easy task. Even if we get the technical aspects of explainable models and post-hoc explanations right, there are still many human factors that must be handled carefully. The so-called [*A-levels Scandal*](#) in the United Kingdom (UK) is an object lesson in failing to understand human factors for high-

risk ML-based decisions. As COVID lockdowns took hold across the United Kingdom in the Spring of 2020, students, teachers and government officials realized that standardized tests could not take place as usual. As a first attempt to remedy the problems with national standardized testing, teachers were asked to estimate student performance on the important A-level exams that determine college entrance and affect other important life outcomes. Unfortunately, teacher estimates were seen as implausibly positive, to the level that using the estimated student performance would be unfair to past and future students.

To address teacher's positive biases, the government Office of Qualifications and Examinations Regulation (Ofqual), decided to implement an algorithm to adjust teacher predictions. The statistical methodology of the adjustment algorithm was implemented by experts and a [model document](#) was released after students received their grades. The algorithm was designed to generate a final distribution of grades that was similar to results in previous years. It preserved teacher rankings, but used past school performance to adjust grades downward. Students in Scotland were first to see the results. In that part of the UK, "35.6 percent of grades were adjusted down by a single grade, while 3.3 percent went down by two grades, and 0.2 went down by three" according to [ZDNet](#).

Over the next few months, student outcry over likely bias against poorer schools and regions of the country caused a massive, slow-burning AI incident. Even though officials had seen the problems in Scotland, they applied the same process in England, but instated a free appeals process and the right for students to re-test at a later date. In the end, irreparable damage to public trust could not be undone, and the transparent but biased notion of adjusting individual scores by past school performance was too much for many to stomach. In the end, the UK government decided to use the original teacher estimates. According to [Wired](#), "the government has essentially passed the administrative buck to universities, who will now have to consider honouring thousands more offers – they have said that despite u-turn, it will not be possible to honour all original offers." The same article also pointed that teacher estimates of student performance had shown racial bias in the past. What a mess.

Shockingly, other institutions also adopted the idea of algorithmic scores for life-changing college entrance exams. The International Baccalaureate (IB) is an elite educational program that offers an advanced uniform curriculum for secondary school students all over the world. In the Spring of 2020, the IB used an algorithm for student scores that was [reported to be](#) "hastily deployed after canceling its usual spring-time exams due to Covid-19. The system used signals including a student's grades on assignments and grades from past grads at their school." Because of the timing, unanticipated negative scores were extremely hurtful to students applying to US colleges and universities, who reserve spaces for IB students based on past performance, but can cancel based on final performances, "shattering their plans for the fall and beyond." Some students algorithmic scores were so low they may have lost placement

in prestigious universities in the US and their safety schools in their home country. What's worse, and unlike the the Ofqual algorithm, the IB was not forthcoming with how their algorithm worked, and appeals came with an almost \$800 price tag.

Putting aside the IB's lack of transparency, there seem to be three major issues at play in these incidents. Scale is an inherent risk of ML, and these algorithms were used on many students across the globe. Large scale translates to high materiality, and transparency alone is not enough to offset issues of trust and bias. Understanding is not trust. Ofquals technical report and other [public analyses](#) were over the heads of many students and parents. But what was not over their heads is that poorer areas have worse public schools and that affected students twice in 2020. Once in an overall manner like every year, and then again when their scores were adjusted downward. Second, was the seriousness of the decision. College admittance plays a huge role in the rest of many people's lives. The serious nature of the decision cranks up the materiality to an even higher degree — possibly to an impossible degree where failure was guaranteed. ML is inherently probabilistic. It will be wrong. And in when the stakes are this high, the public just might not accept it.

The third major issue at play here is the clear nature of disparate impact. For example, very small classes were not scored with the algorithm. Where are there the most very small classes? Private schools. A [Verge](#) article claims that “fee-paying private schools (also known as *independent schools*) disproportionately benefited from the algorithm used. These schools saw the amount of grades A and above increase by 4.7 percent compared to last year.” [ZDNet](#) reported that the “pass rate for students undertaking higher courses in deprived locations across Scotland was reduced by 15.2%, in comparison to 6.9% in more affluent areas.” Adjusting by postal code or past school performance bakes in systemic biases, and students and parents understood this at an emotional level. As quoted in the [BBC](#), Scotland’s Education Secretary realized in the end that the debacle left “young people feeling their future had been determined by statistical modeling rather than their own ability.” We should think about how we would feel if this incident had affected us or our children. Despite all the hype around automated-decision making, almost no one wants to feel that their future is set by an algorithm.

Although this may have been a doomed an impossibly high-materiality application of ML from the beginning, more could have been done to increase public trust. For example, Ofqual could have published the algorithm before applying it to students. They also could have taken public feedback on the algorithm before using it. Jenni Tennison, a UK-based open data advocate notes, “Part of the problem here is that these issues came out only after the grades were given to students, when we could have been having these discussions and being examining the algorithm and understanding the implications of it much, much earlier.” The take home lessons here are that technical transparency is not the same as broad social understanding, and that understanding — if it can even be achieved, does not guarantee trust. Even if we've

done a good job on technical transparency, as put forward in this chapter, there is still a great deal of work that must be done to ensure an ML system works as expected for users, or subjects. Finally this is just one AI incident, and although it's a big one, there are smaller ones harming people right now and more people will be harmed by AI systems in the future. As Ms. Tennison put it, "This has hit the headlines, because it affects so many people across the country, and it affects people who have a voice. There's other automated decision making that goes on all the time, around benefits, for example, that affect lots of people who don't have this strong voice."

Resources

- *An Introduction to Machine Learning Interpretability*
- *Designing Inherently Interpretable Machine Learning Models*
- *Explanatory Model Analysis*
- *General Pitfalls of Model-Agnostic Interpretation Methods for Machine Learning Models*
- *Interpretable Machine Learning*
- *Limitations of Interpretable Machine Learning Methods*
- *On the Art and Science of Explainable Machine Learning*
- When Not to Trust Explanations

Debugging Machine Learning Systems for Safety and Performance

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

For decades, error or accuracy on hold-out test data has been the standard by which machine learning (ML) models are judged. Unfortunately, as machine learning (ML) models are embedded into artificial intelligence (AI) systems that are deployed more broadly and for more sensitive applications, the standard approaches for ML model assessment have proven inadequate. For instance, overall test data area under the curve (AUC) tells us almost nothing about bias and algorithmic discrimination, lack of transparency, privacy harms, or security vulnerabilities. Yet, these problems are often why AI systems fail once deployed. For acceptable *in vivo* performance, we simply must push beyond traditional *in silica* assessments designed primarily for research prototypes. Moreover, the best results for safety and performance occur when organizations are able to mix and match appropriate cultural competencies and process controls from Chapter 1 with ML technology that promotes trust. Chapter 3 presents sections on training, debugging, and deploying ML systems that delve into

the numerous technical approaches for testing and improving *in vivo* safety, performance, and trust in AI.



NIST does not review, approve, condone or otherwise address any content in this book, including any claims relating to the AI RMF. All AI RMF content is simply the authors' opinions and in no way reflects an official position of NIST or any official or unofficial relationship between NIST and the book or the authors.

NIST AI RMF Crosswalk

Chapter Section	NIST AI RMF Sub-categories
Reproducibility	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Data Quality	GOVERN_1.2, MAP_2.3, MAP_4.2, MEASURE_2.1
Benchmarks and Alternatives	GOVERN_1.3, MANAGE_2.1
Calibration	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Construct Validity	MAP_2.3
Assumptions and Limitations	GOVERN_1.3, GOVERN_6.1, MAP_2.1
Default Loss Functions	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Multiple Comparisons	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
The Future of Safe and Robust Machine Learning	MEASURE_2.5
Software Testing	GOVERN_2.1, GOVERN_4.3, GOVERN_6.1, MAP_4, MEASURE_1.3
Traditional Model Assessment	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MAP_4.1, MEASURE_1.2, MEASURE_1.3, MEASURE_2.1
Distribution Shifts	GOVERN_1.2, GOVERN_1.3, GOVERN_1.4, MANAGE_3.1, MANAGE_4.1
Epistemic Uncertainty and Data Sparsity	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Instability	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Leakage	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Looped Inputs	GOVERN_1.3, MAP_1.7, MAP_2.2,

Chapter Section	NIST AI RMF Sub-categories
Overfitting	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Shortcut Learning	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Underfitting	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Underspecification	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Residual Analysis	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MAP_3.2, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Sensitivity Analysis	GOVERN_1.2, GOVERN_1.3, MAP_2.3, MAP_3.2, MAP_5.2, MEASURE_1, MEASURE_2.1, MEASURE_2.3
Benchmark Models	GOVERN_1.3, MANAGE_2.1
Remediation: Fixing Bugs	GOVERN, MAP, MANAGE
Domain Safety	GOVERN_3.1, GOVERN_4.1, GOVERN_5, MAP_1.2, MAP_1.7, MAP_2.3, MAP_3.1, MAP_5, MEASURE_1.3, MEASURE_2.4, MEASURE_2.5, MEASURE_3, MEASURE_4
Model Monitoring	GOVERN_1.2, GOVERN_1.3, GOVERN_1.4, MAP_5.1, MEASURE_2.5, MANAGE_3, MANAGE_4

Applicable AI trustworthiness characteristics include: Safe, Secure_and_Resilient, Valid_and_Reliable

Training

The discussion of training ML algorithms begins with reproducibility, because without that, it's impossible to know if any one version of an ML system is really any better than another. Data and feature engineering will be addressed briefly and the training section closes by outlining key points for model specification.

Reproducibility

Without reproducibility, we're building on sand. Reproducibility is fundamental to all scientific efforts, including AI. Without reproducible results, it's very hard to know if day-to-day efforts are improving, or even changing, an ML system. Reproducibility helps ensure proper implementation and testing, and some customers may simply demand it. The techniques discussed below are some of the most common that data scientists and ML engineers use to establish a solid, reproducible foundation for their ML systems.

- **Benchmark Models:** Benchmark models are important safety and performance tools for training, debugging, and deploying ML systems. They'll be addressed several times in Chapter 3. In the context of model training and reproducibility, we should always build from a reproducible benchmark model. This allows a checkpoint for rollbacks if reproducibility is lost, but it also enables real progress. If yesterday's benchmark is reproducible, and today's gains above and beyond that benchmark are also reproducible, that's real and measurable progress. If system performance metrics bounce around before changes are made, and they're still bouncing around after changes are made, we have no idea if our changes helped or hurt.
- **Hardware:** As ML systems often leverage hardware acceleration via graphical processing units (GPUs) and other specialized system components, hardware is still of special interest for preserving reproducibility. If possible, try to keep hardware as similar as possible across development, testing, and deployment systems.
- **Environments:** ML systems always operate in some computational environment, specified by the system hardware, system software, and our data and ML software stack. Changes in any of these can affect the reproducibility of ML outcomes. Thankfully, tools like Python virtual environments and Docker containers that preserve software environments have become commonplace in the practice of data science. Additional specialized environment management software from [Domino Data Labs](#), [gigantum](#), [TensorFlow TFX](#), and [Kubeflow](#) can provide even more expansive control of computational environments.
- **Metadata:** Data about data is essential for reproducibility. Track all artifacts associated with the model, e.g., datasets, preprocessing steps, model, data and model validation results, human sign offs, and deployment details. Not only does this allow rolling-back to a specific version of a dataset or model, but it also allows for detailed debugging and forensic investigations of AI incidents. For an open-source example of a nice tool for tracking metadata, checkout [TensorFlow ML Metadata](#).
- **Random Seeds:** Set by data scientists and engineers in specific code blocks, random seeds are the plow-horse of ML reproducibility. Unfortunately, they often come with language- or package-specific instructions. Seeds can take some time learn in different software, but when combined with careful testing, random seeds enable the building blocks of intricate and complex MLsystems to retain reproducibility. This is a prerequisite for overall reproducibility.
- **Version Control:** Minor code changes can lead to drastic changes in ML results. Changes to our own code plus it's dependencies must be tracked in a professional version control tool for any hope of reproducibility. Git and GitHub are free and ubiquitous resources for software version control, but there are plenty of other

options to explore. Crucially, data can also be version-controlled with tools like [Pachyderm](#) and [DVC](#), enabling traceability in changes to data resources.

Though it may take some experimentation, some combination of these approaches and technologies should work to assure a level of reproducibility in our ML systems. Once this fundamental safety and performance control is in place, it's time to consider other baseline factors like data quality and feature engineering.



Several topics like benchmarks, anomaly detection and monitoring are ubiquitous in model debugging and ML safety and they appear in several different sections and contexts in [Chapter 3](#).

Data Quality

Entire books have been written about data quality and feature engineering for ML and MLsystems. This short subsection highlights some of the most critical aspects of this vast practice area from a safety and performance perspective. First and foremost, biases, confounding features, incompleteness and noise in development data form important assumptions and define limitations of our models. Other basics, like size and shape of a dataset and are important safety and performance considerations. ML algorithms are hungry for data. Both small data and wide, sparse data can lead to catastrophic performance failures in the real-world, because both give rise to scenarios in which system performance appears normal on test data, but is simply untethered to real-world phenomena. Small data can make it hard to detect underfitting, underspecification, overfitting, or other fundamental performance problems. Sparse data can lead to over-confident predictions for certain input values. If an ML algorithm did not see certain data ranges during training due to sparsity issues, most ML algorithms will issue a predictions in those ranges with no warning that the prediction is based on almost nothing. Fast-forwarding to our chapter case discussion, there is simply not enough training video in the world to fill out the entire space of examples that self-driving cars need to learn to drive. For example, people crossing the road at night with a bicycle is a danger most humans will recognize, but without many frames of labeled video of this somewhat rare event, a deep learning system's ability to handle this situation will likely be compromised due to sparsity in training data.

A number of other data problems can cause safety worries, such as poor data quality leading to entanglement or misrepresentation of important information and overfitting, or ML data and model pipeline problems. In [Chapter 3](#), entanglement means features, entities, or phenomenon in training data proxying for other information with more direct relationships to the target. (E.g., snow proxying for a Huskie in object recognition.) Overfitting refers to memorizing noise in training data and

resulting optimistic error estimates, and pipeline issues are problems that arise from combining different stages of data preparation and modeling components into one prediction-generating executable. Table [Table 3-1](#) below can be applied to most standard ML data to help identify common data quality problems with safety and performance implications.

Table 3-1. Table of common data quality problems with symptoms and proposed solutions. Adapted from The George Washington University DNSC 6314 (Machine Learning I) class notes with permission.

Problem	Common Symptoms	Possible Solutions
Biased data: When a data set contains information about the phenomenon of interest, but that information is consistently and systematically wrong. (See Chapter 4 for more information.)	Biased models and biased, dangerous or inaccurate results. Perpetuation of past social biases and discrimination.	Consult with domain experts and stakeholders. Apply the scientific method and design of experiment approaches. (Get more data. Get better data.)
Character data: When certain columns, features or instances are represented with strings of characters instead of numeric values.	Information loss. Biased models and biased, dangerous or inaccurate results. Long, intolerable training times.	Various numeric encoding approaches (e.g., label encoding, target or feature encoding). Appropriate algorithm selection, e.g. Tree-based models, naïve Bayes classification.
Data leakage: When information from validation or test partitions leaks into training data.	Unreliable or dangerous out-of-domain predictions. Over-fit models and inaccurate results. Overly optimistic <i>in silico</i> performance estimates.	Data governance. Ensuring all dates in training are earlier than in validation and test. Ensuring identical identifiers do not occur across partitions. Careful application of feature engineering — engineer after partitioning, not before.
Dirty data: A combination of all the issues in this table, very common in real-world datasets.	Information loss. Biased models and biased, inaccurate results. Long, intolerable training times. Unstable and unreliable parameter estimates and rule generation. Unreliable or dangerous out-of-domain predictions.	Combination of solution strategies herein.
Disparate feature scales: When features, such as age and income, are recorded on different scales.	Unreliable parameter estimates, biased models, and biased, inaccurate results.	Standardization. Appropriate algorithm selection, e.g. tree-based models.
Duplicate data: Rows, instances, or entities that occur more than intended.	Biased results due to unintentional overweighing of identical entities during training. Biased models, and biased, inaccurate results.	Careful data cleaning in consultation with domain experts.
Entanglement: When features, entities, or phenomenon in training data proxy for other information with more direct relationships to the target. (E.g., snow proxying for a Huskie in object recognition.)	Unreliable or dangerous out-of-domain predictions. Shortcut learning.	Apply the scientific method and design of experiment approaches. Apply interpretable models and post-hoc explanation. In-domain testing.

Problem	Common Symptoms	Possible Solutions
Fake or poisoned data: Data, features, attributes, phenomenon or entities that are injected into or manipulated in training data to elicit artificial model outcomes.	Unreliable or dangerous out-of-domain predictions. Biased models and biased, inaccurate results.	Data governance. Data security. Application of Robust ML approaches.
High cardinality categorical features: Features such as postal codes or product identifiers that represent many categorical levels of the same attribute.	Over-fit models and inaccurate results. Long, intolerable compute times. Unreliable or dangerous out-of-domain predictions.	Target or feature encoding or variants average-, median, BLUP-by-level. Discretization. Embedding approaches, e.g. entity embedding neural networks, factorization machines.
Imbalanced target: When one target class or value is much more common than others.	Single class model predictions. Biased model predictions.	Proportional over- or under-sampling. Inverse prior probability weighting. Mixture models, e.g. zero-inflated regression methods. Post-hoc adjustment of predictions or decision thresholds.
Incomplete data: When a dataset does not encode information about the phenomenon of interest. When uncollected information confounds model results.	Useless models, meaningless, or dangerous results.	Consult with domain experts and stakeholders. Apply the scientific method and design of experiment approaches. (Get more data. Get better data.)
Missing values: When specific rows or instances are missing information.	Information loss. Biased models and biased, inaccurate results.	Imputation. Discretization (i.e., binning). Appropriate algorithm selection, e.g. tree-based models, naïve Bayes classification.
Noise: Data that fails to encode clear signals for modeling. Data with the same input values and different target values.	Unreliable or dangerous out-of-domain predictions. Poor performance during training	Consult with domain experts and stakeholders. Apply the scientific method and design of experiment approaches. (Get more data. Get better data.)
Non-normalized data: Data in which values for the same entity are represented in different ways.	Unreliable out-of-domain predictions. Long, intolerable training times. Unreliable parameter estimates and rule generation.	Careful data cleaning in consultation with domain experts.
Outliers: Rows or instances of data that are strange or unlike others.	Biased models and biased, inaccurate results. Unreliable parameter estimates and rule generation. Unreliable out-of-domain predictions.	Discretization (i.e., binning). Winsorizing. Robust loss functions, e.g. Huber loss functions.
Sparse data: Data with many zeros or missing values; data that does not encode enough information about the phenomenon of interest.	Long, intolerable training times. Meaningless or dangerous results due to lack of information, curse of dimensionality, or model misspecification.	Feature extraction or matrix factorization approaches. Appropriate data representation (i.e., COO, CSR). Application of business rules, model assertions, and constraints to make up for illogical model behavior learned in sparse regions of training data.
Strong multicollinearity (correlation): When features have strong linear dependencies on one another.	Unstable parameter estimates, unstable rule generation, and dangerous or unstable predictions.	Feature selection. Feature extraction. L2 Regularization.

Problem	Common Symptoms	Possible Solutions
Unrecognized time and date formats: Time and date formats, of which there are many, that are encoded improperly by data handling or modeling software.	Unreliable or dangerous out-of-domain predictions. Unreliable parameter estimates and rule generation. Over-fit models and inaccurate results. Overly optimistic <i>in silico</i> performance estimates.	Careful data cleaning in consultation with domain experts.
Wide data: Data with many more columns, features, pixels, or tokens than rows, instances, images or documents. $P \gg N$.	Long, intolerable training times. Meaningless or dangerous results due to curse of dimensionality or model misspecification.	Feature selection, feature extraction, L1 Regularization, models that do not assume $N \gg P$.

There is a lot that can go wrong with data that then leads to unreliable or dangerous model performance in high-risk applications. It might be tempting to think we can feature engineer our way out of data quality problems. But feature engineering is only as good as the thought and code used to perform it. If we're not extremely careful with feature engineering, we're likely just creating more bugs and complexity for ourselves. Common issues with feature engineering in ML pipelines include:

- API or version mismatches between data cleaning, preprocessing and inference packages.
- Failing to apply all data cleaning and transformation steps during inference.
- Failing to re-adjusting for over- or under-sampling during inference.
- Inability to handle values unseen during training gracefully or safely during inference.

Of course, many other problems can arise in data preparation, feature engineering, and associated pipelines, especially as the types of data that ML algorithms can accept for training becomes more varied. Tools that detect and address such problems are also an important part of the data science toolkit. For Python Pandas users, the [pandas-profiling](#) tool is a visual aide that helps to detect many basic data quality problems. R users also have options, as discussed by Mateusz Staniak and Przemysław Biecek in [The Landscape of R Packages for Automated Exploratory Data Analysis](#).

Model Specification for Real-world Outcomes

Once our data preparation and feature engineering pipeline is hardened, it's time to think about ML model specification. Considerations for real-world performance and safety are quite different from getting published or maximizing performance on ML contest leaderboards. While measurement of validation and test error remain important, bigger questions of accurately representing data and commonsense real-world phenomena have the highest priority. This subsection address model specification for safety and performance by highlighting the importance of benchmarks and alternative models, calibration, construct validity, assumptions and limitations, proper loss

functions, avoiding multiple comparisons and by previewing the emergent disciplines of robust ML and ML safety and reliability.

Benchmarks and Alternatives

When starting a ML modeling task, it's best to begin with a peer-reviewed training algorithm, and ideally to replicate any benchmarks associated with that algorithm. While academic algorithms rarely meet all the needs of complex business problems, starting from a well-known algorithm and benchmarks provides a baseline assurance that the training algorithm is implemented correctly. Once this sanity check is addressed, then think about tweaking a complex algorithm to address specific quirks of a given problem.

Along with comparison to benchmarks, evaluation of numerous alternative algorithmic approaches is another best practice that can improve safety and performance outcomes. The exercise of training many different algorithms and judiciously selecting the best of many options for final deployment typically results in higher-quality models because it increases the number of models evaluated and forces users to understand differences between them. Moreover, evaluation of alternative approaches is important in complying with a broad set of U.S. non-discrimination and negligence standards. In general, these standards require evidence that different technical options were evaluated and an appropriate trade-off between consumer protection and business need was made before deployment.

Calibration

Just because a number between 0 and 1 pops out the end of a complex ML pipeline does not make it a probability. The uncalibrated probabilities generated by most ML classifiers usually have to be post-processed to have any real meaning as probabilities. We typically use a scaling process, or even another model, to ensure that when a pipeline outputs 0.5, that the event in question actually happened to about 50% of similar entities in past recorded data. [Scikit learn](#) provides some basic diagnostics and functions for ML classifier calibration. Calibration issues can affect regression models too, when the distribution of model outputs don't match the distribution of known outcomes. For instance, many numeric quantities in insurance are not normally distributed. Using a default squared loss function, instead of loss functions from the gamma or Tweedie family, may result in predictions that are not distributed like values from the known underlying data-generating process. However we think of calibration, the fundamental issue is that ML model predictions don't match to reality. We'll never make good predictions and decisions like this. We need our probabilities to be aligned to past outcome rates and we need our regression models to generate predictions of the same distribution as the modeled data-generating process.

Construct Validity

Construct validity is an idea from social science, and in particular psychometrics and testing. Construct validity means that there is a reasonable scientific basis to believe that test performance is indicative of the intended construct. Said another way, is there any scientific evidence that the questions and scores from a standardized test can predict college or job performance? Why are we bringing this up in an ML book? Because ML models are often used for the same purposes as psychometric tests these days, and in our opinion, ML models often lack construct validity. Worse, ML algorithms that don't align with fundamental structures in training data or in their real-world domain can cause serious incidents.

Consider the choice between an ML model and a linear model, where many of us simply default to using an ML model. Selecting an ML algorithm for a modeling problem comes with a lot of basic assumptions — essentially that high-degree interactions and nonlinearity in input features are important drivers of the predicted phenomenon. Conversely, choosing to use a linear model implicitly downplays interactions and nonlinearities. If those qualities are important for good predictions, they'll have to be specified explicitly for the linear model. In either case, it's important to take stock of how main effects, correlations and local dependencies, interactions, nonlinearities, clusters, outliers, and hierarchies in training data, or in reality, will be handled by a modeling algorithm and to test those mechanisms. For optimal safety and performance once deployed, dependencies on time, geographical locations, or connections between entities in various types of networks must also be represented within ML models. Without these clear links to reality, ML models lack construct validity and are unlikely to exhibit good *in vivo* performance. Feature engineering, constraints, loss functions, model architectures and other mechanisms can all be used to match a model to its task.

Assumptions and Limitations

Biases, entanglement, incompleteness, noise, ranges, sparsity, and other basic characteristics of training data begin to define the assumptions and limitations of our models. As discussed directly above, modeling algorithms and architectures also carry assumptions and limitations. For examples, tree-based models usually can't extrapolate beyond ranges in training data. Hyperparameters for ML algorithms are yet another place where hidden assumptions can cause safety and performance problems. Hyperparameters can be selected based on domain knowledge or via technical approaches like grid search and Bayesian optimization. The key is not settling for defaults, choosing settings systematically, and without tricking ourselves due to multiple comparison issues. Testing for independence of errors between rows and features in training data or plotting model residuals and looking for strong patterns are general and time-tested methods for ensuring some basic assumptions have been addressed. It's unlikely we'll ever circumvent all the assumptions and limitations of

our data and model. So we need to document any unaddressed or suspected assumptions and limitations in model documentation, and ensure users understand what uses of the model could violate its assumptions and limitations. Those would be considered out-of-scope or *off-label* uses — just like using a prescription drug in improper ways. By the way, construct validity is linked to model documentation and risk management frameworks focusing on model limitations and assumptions. Oversight professionals want practitioners to work through the hypothesis behind their model in writing, and make sure it's underpinned by valid constructs and not assumptions.

Default Loss Functions

Another often-unstated assumption that comes with many learning algorithms involves squared loss functions. Many ML algorithms use a squared loss function by default. In most instances, a squared loss function, being additive across observations and having a linear derivative, is more a matter of mathematical convenience than anything else. With modern tools such as `autograd`, this convenience is increasingly unnecessary. We should match our choice of loss function with our problem domain.

Multiple Comparisons

Model selection in ML often means trying many different sets of input features, model hyperparameters, and other model settings such as probability cutoff thresholds. We often use stepwise feature selection, grid searches, or other methods that try many different settings on the same set of validation or holdout data. Statisticians might call this a *multiple comparisons* problem and likely point out the more comparisons we do, the likelier we are to stumble upon some settings that simply happen to look good in our validation or holdout set. This is a sneaky kind of overfitting where we reuse the same holdout data too many times, select features, hyperparameters, or other settings that work well there, and then experience poor *in vivo* performance later on. Hence, `re-usable holdout` approaches, that alter or resample validation or holdout data to make our feature, hyperparameter, or other settings more generalizable.

The Future of Safe and Robust Machine Learning

The new field of `robust ML` is churning out new algorithms with improved stability and security characteristics. Various researchers are creating new learning algorithms with guarantees for optimality, like `optimal sparse decision trees`. And researchers have put together excellent `tutorial materials` on ML safety and reliability. Today, these approaches require custom implementations and extra work, but hopefully soon these safety and performance advances will be more widely available soon.

Model Debugging

Once a model has been properly specified and trained, the next step in the technical safety and performance assurance process is testing and debugging. In years past, such assessments focused on aggregate quality and error rates in holdout data. As ML models are incorporated in public-facing MLsystems, and the number of publicly reported AI incidents is increasing dramatically, it's clear that more rigorous validation is required. The new field of **model debugging** is rising to meet this need. Model debugging treats ML models more like code and less like abstract mathematics. It applies a number of different testing methods to find software flaws, logical errors, inaccuracies, and security vulnerabilities in ML models and MLsystem pipelines. Of course, these bugs must also be fixed when they are found. This section explores model debugging in some detail, starting with basic and traditional approaches, then outlining the common bugs we're trying to find, moving onto specialized testing techniques, and closing with a discussion of bug remediation methods.



In addition to many explainable ML models, the open source package **PiML** contains an exhaustive set of debugging tools for ML models trained on structured data. Even if it's not an exact fit a given use case, it's a great place to learn more and gain inspiration for model debugging.

Software Testing

Basic software testing becomes much more important when we stop thinking of pretty figures and impressive tables of results as the end goal of an ML model training task. When MLsystems are deployed, they need to work correctly under various circumstances. Almost more than anything else related to ML systems, making software work is an exact science. Best practices for software testing are well-known and can even be made automatic in many cases. At a minimum, mission-critical ML systems should undergo:

- **Unit testing:** All functions, method, subroutines or other code blocks should have tests associated with them to ensure they behave as expected, accurately, and are reproducible. This ensures the building blocks of an ML system are solid.
- **Integration testing:** All APIs and interfaces between modules, tiers, or other subsystems should be tested to ensure proper communication. API mismatches after backend code changes are a classic failure mode for ML systems. Use integration testing to catch this and other integration fails.
- **Functional testing:** Functional testing should be applied to ML system user interfaces and endpoints to ensure that they behave as expected once deployed.

- **Chaos testing:** Testing under chaotic and adversarial conditions can lead to better outcomes when our ML systems faces complex and surprising *in vivo* scenarios. Because it can be difficult to predict all the ways an ML systems can fail, chaos testing can help probe a broader class of failure modes, and provide some cover against so-called “unknown unknowns.”

Two additional ML-specific tests should be added into the mix to increase quality further:

- **Random attack:** Random attacks are one way to do chaos testing in ML. Random attacks expose ML models to vast amounts of random data to catch both software and math problems. The real world is a chaotic place. Our ML system will encounter data for which it's not prepared. Random attacks can decrease those occurrences and any associated glitches or incidents.
- **Benchmarking:** Use benchmarks to track system improvements over time. ML systems can be incredibly complex. How can we know if the 3 lines of code an engineer changes today made a difference in the performance of the system as a whole? If system performance is reproducible, and benchmarked before and after changes, it's much easier to answer such questions.

ML is software. So, all the testing that's done on traditional enterprise software assets should be done on important ML systems as well. If we don't know where to start with modeling debugging, we start with random attacks. Readers may be shocked at the math or software bugs random data can expose in ML systems. When we can add benchmarks to our organization's continuous integration/continuous development (CI/CD) pipelines, that's the another big step toward assuring the safety and performance of ML systems.



Random attacks are probably the easiest and most effective way to get started with model debugging. If debugging feels overwhelming, or you don't know where to start, start with random attacks.

Traditional Model Assessment

Once we feel confident that the code in our ML systems is functioning as expected, it's easier to concentrate on testing the math of our ML algorithms. Looking at standard performance metrics is important. But it's not the end of the validation and debugging process — it's the beginning. While exact values and decimal points matter, from a safety and performance standpoint, they matter much less than they do on the leaderboard of an ML contest. When considering in-domain performance, it's less about exact numeric values of assessment statistics, and more about mapping *in silico* performance to *in vivo* performance.

If possible, try to select assessment statistics that have a logical interpretation and practical or statistical thresholds. For instance, root mean squared error (RMSE) can be calculated for many types of prediction problems, and crucially, it can be interpreted in units of the target. Area under the curve (AUC), for classification tasks, is bounded between 0.5 at the low end and 1.0 at the high end. Such assessment measures allow for commonsense interpretation of ML model performance and for comparison to widely accepted thresholds for determining quality. It's also important to use more than one metric and to analyze performance metrics across important segments in our data and across training, validation, and testing data partitions. When comparing performance across segments within training data, it's important that all those segments exhibit roughly equivalent and high quality performance. Amazing performance on one large customer segment, and poor performance on everyone else, will look fine in average assessment statistic values like RMSE. But, it won't look fine if it leads to public brand damage due to many unhappy customers. Varying performance across segments can also be a sign of underspecification, a serious ML bug we'll dig into below. Performance across training, validation and test datasets are usually analyzed for under and overfitting too. Like model performance, we can look for over and underfitting across entire data partitions or across segments.

Another practical consideration related to traditional model assessment is selecting a probability cutoff threshold. Most ML models for classification generate numeric probabilities, not discrete decisions. Selecting the numeric probability cutoff to associate with actual decisions can be done in various ways. While it's always tempting to maximize some sophisticated assessment measure, it's also a good idea to consider real-world impact. Let's consider a classic lending example. Say a probability of default model threshold is originally set at 0.15, meaning that everyone who scores less than a 0.15 probability of default is approved for a loan, and those that score at the threshold or over are denied. Think through questions like:

- What is the expected monetary return for this threshold? What is the financial risk?
- How many people will get the loan at this threshold?
- How many women? How many minority group members?

Outside of the probability cutoff thresholds, it's always a good idea to estimate in-domain performance, because that's what we really care about. Assessment measures are nice, but what matters is making money versus losing money, or even saving lives versus taking lives. We can take a first crack at understanding real-world value by assigning monetary, or other, values to each cell of a confusion matrix for classification problems or to each residual unit for regression problems. Do a back-of-the-napkin calculation. Does it look like our model will make money or lose money? Once we get the gist of this kind of valuation, we can even incorporate value levels for

different model outcomes directly into ML loss functions, and optimize towards the best-suited model for real-world deployment.

Error and accuracy metrics will always be important for ML. But once ML algorithms are used in deployed ML systems, numeric values and comparisons matter less than they do for publishing papers and data science competitions. So, keep using traditional assessment measures, but try to map them to in-domain safety and performance.

Machine Learning Bugs

We've discussed a lack of reproducibility, data quality problems, proper model specification, software bugs and traditional assessment. But there's still a lot more that can go wrong with complex ML systems. When it comes to the math of ML, there are a few emergent gotchas and many well-known pitfalls. This subsection will discuss some usual suspects and some dark horse bugs including distributional shifts, epistemic uncertainty, weakspots, instability, leakage, looped inputs, overfitting, shortcut learning, underfitting, and underspecification.

Vocabulary Relating to Reliability, Robustness and Resilience

Under the NIST AI Risk Management Framework *resilience* is a synonym for *security*, and *robustness* and *reliability* are used somewhat synonymously. Traditionally, and according to [ISO/IEC TS 5723:2022](#), reliability is related to the “ability of an item to perform as required, without failure, for a given time interval, under given conditions.” In ML and statistics, reliability is often related to performance uncertainty and may be measured with confidence intervals, control limits, or conformal prediction techniques. ISO/IEC TS 5723:2022 relates robustness to generalization and defines it as “the ability of an item to maintain its level of performance under a variety of circumstances.” In ML and statistics, a model’s robustness might be tested under covariate perturbation (or *distribution shift*). Robustness has also come to be associated with **robust ML** — or the study of preventing adversaries from manipulating models with adversarial examples and data poisoning.

A lack of robustness and reliability may arise from many of the bugs discussed in this chapter and book. In this chapter, we have used terms like *instability*, distribution shift and related ML vocabulary (e.g., *data leakage*, *epistemic uncertainty*, *underspecification*) to describe issues relating to a lack of robustness and reliability. Chapter [Chapter 5](#) addresses security for ML systems.

Distribution Shifts

Shifts in the underlying data between different training data partitions and after model deployment are common failure modes for ML systems. Whether it's a new

competitor entering a market or a devastating world-wide pandemic, the world is a dynamic place. Unfortunately most of today's ML systems learn patterns from static snapshots of training data and try to apply those patterns in new data. Sometimes that data is holdout validation or testing partitions. Sometimes it's live data in a production scoring queue. Regardless, drifting distributions of input features is a serious bug that must be caught and squashed.



Systems based on adaptive, online, or reinforcement learning, or that update themselves with minimal human intervention, are subject to serious adversarial manipulation, error propagation, feedback loop, reliability, and robustness risks. While these systems may represent the current state of the art, they need high-levels of risk management.

When training ML models, watch out for distributional shifts between training, cross-validation, validation, or test sets using population stability index (PSI), KS-, t-, or other appropriate measures. If a feature has a different distribution from training partition to another, drop it or regularize it heavily. Another smart test for distributional shifts to conduct during debugging is to simulate distributional shifts for potential deployment conditions and re-measure model quality, with a special focus for poor-performing rows. If we're worried about how our model will perform during a recession, we can simulate distributional shifts to simulate more late payments, lower cash flow, and higher credit balances and then see how our model performs. It's also crucial to record information about distributions in training data so that drift after deployment can be detected easily.

Epistemic Uncertainty and Data Sparsity

Epistemic uncertainty is a fancy way of saying instability and errors that arise from a lack of knowledge. In ML, models traditionally gain knowledge from training data. If there are parts of our large multidimensional training data that are sparse, it's likely our model will have a high degree of uncertainty in that region. Sound theoretical and far-fetched? It's not. Consider a basic credit lending model. We tend to have lots of data about people who already have credit cards and pay their bills, and tend to lack data on people who don't have credit cards (they're past credit card data doesn't exist) or don't pay their bills (because the vast majority of customers pay). It's easy to know to extend credit cards to people with high credit scores that pay their bills. The hard decisions are about people with shorter or bumpier credit histories. The lack of data for the people we really need to know about can lead to serious epistemic uncertainty issues. If only a handful of customers, out of millions, are 4-5 months late on their most recent payment then an ML model simply doesn't learn very much about the best way to handle these people.

This phenomenon is visible in [Link to Come], where the example model is nonsensical for people who are more than two months late on their most recent payment. This region of poor, and likely unstable, performance is sometimes known as a *weak spot*. It's hard to find these weak spots by looking at aggregate error or performance measures. This is just one reason of many to test models carefully over segments in training or holdout data. It's also why we pair partial dependence and individual conditional expectation plots with histograms in [Chapter 2](#). In these plots we can see if model behavior is supported by training data, or not. Once we've identified a sparse region of data leading to epistemic uncertainty and weakspots, we usually have to turn to human knowledge — by constraining the form of the model to behave logically based on domain experience, augmenting the model with business rules, or potentially handing the cases that fall into sparse regions over to human workers to make tough calls.

Instability

ML models can exhibit instability, or lack of robustness or reliability, in the training process or when making predictions on live data. Instability in training is often related to small training data, sparse regions of training data, highly correlated features within training data, or high-variance model forms, such as deep single decision trees. Cross-validation is a typical tool for detecting instability during training. If a model displays noticeably different error or accuracy properties across cross-validation folds then we have an instability problem. Training instability can often be remediated with better data and lower-variance model forms such as decision tree ensembles. Plots of ALE or ICE also tend to reveal prediction instability in sparse regions of training data, and instability in predictions can be analyzed using sensitivity analysis: perturbations, simulations, stress-testing, and adversarial example searches.



There are two easy ways to think about instability in ML:

1. When a small change to input data results in a large change in output data.
2. When the addition of a small amount of training data results in a largely different model upon retraining.

If probing our response surface or decision boundary with these techniques uncovers wild swings in predictions, or our ALE or ICE curves are bouncing around, especially in high or low ranges of feature values, we also have an instability problem. This type of instability can often be fixed with constraints and regularization. Check out the code examples in Chapter 8 to see this remediation in action.

Leakage

Information leakage between training, validation, and test data partitions happens when information from validation and testing partitions leaks into a training partition, resulting in overly optimistic error and accuracy measurements. Leakage can happen for a variety of reasons, including:

- **Feature Engineering:** If used incorrectly, certain feature engineering techniques such as imputation or principal components analysis (PCA) may contaminate training data with information from validation and test data. To avoid this kind of leakage, perform feature engineering uniformly, but separately, across training data partitions. Or ensure that information, like means and modes used for imputation, are calculated in training data and applied to validation and testing data, and not vice-versa.
- **Mistreatment of Temporal Data:** Don't use the future to predict the past. Most data has some association with time, whether explicit as in time-series data, or some other implicit relationship. Mistreating or breaking this relationship with random sampling is a common cause of leakage. If we're dealing with data where time plays a role, time needs to be used in constructing model validation schemes. The most basic rule is that the earliest data should be in training partitions while later data should be divided into validation and test partitions, also according to time. A solid (and free) resource for time-series forecasting best practices is the text *Forecasting Principles and Practice*.
- **Multiple Identical Entities:** Sometimes the same person, financial or computing transaction, or other modeled entity will be in multiple training data partitions. When this occurs, care should be taken to ensure that ML models do not memorize characteristics of these individuals then apply those individual-specific patterns to different entities in new data.

Keeping an untouched, time-aware holdout set for an honest estimate of real-world performance can help with many of these different leakage bugs. If error or accuracy on such a holdout set looks a lot less rosy than on partitions used in model development, we might have a leakage problem. More complex modeling schemes involving stacking, gates, or bandits can make leakage much harder to prevent and detect. However, a basic rule of thumb still applies: do not use data involved in learning or model selection to make realistic performance assessments. Using stacking, gates, or bandits means we need more holdout data for the different stages of these complex models to make an accurate guess at *in vivo* quality. More general controls such as careful documentation of data validation schemes and model monitoring in deployment are also necessary for any ML system.

Looped Inputs

As ML systems are incorporated into broader digitalization efforts, or implemented as part of larger decision support efforts, multiple data-driven systems often interact. In these cases, error propagation and feedback loop bugs can occur. Error propagation occurs when small errors in one system cause or amplify errors in another system. Feedback loops are a way an ML system can fail by being right. Feedback loops occur when an ML system affects its environment and then those effects are re-incorporated into system training data. Examples of feedback loops include when predictive policing leads to over-policing of certain neighborhoods or when employment algorithms intensify diversity problems in hiring by continually recommending correct, but non-diverse, candidates. Dependencies between systems must be documented and deployed models must be monitored so that debugging efforts can detect error propagation or feedback loop bugs.

Overfitting

Overfitting happens when a complex ML algorithm memorizes too much specific information from training data, but does not learn enough generalizable concepts to be useful once deployed. Overfitting is often caused by high-variance models, or models that are too complex for the data at hand. Overfitting usually manifests in much better performance on training data than on validation, cross-validation, and test data partitions. Since overfitting is a ubiquitous problem, there are many possible solutions, but most involve decreasing the variance in our chosen model. Examples of these solutions include:

- **Ensemble Models:** Ensemble techniques, particularly bootstrap aggregation (i.e., bagging) and gradient boosting are known to reduce error from single high-variance models. So, we try one of these ensembling approaches if we encounter overfitting. Just keep in mind that when switching from one model to many, we can decrease overfitting and instability but we'll also likely lose interpretability.
- **Reducing Architectural Complexity:** Neural networks can have too many hidden layers or hidden units. Ensemble models can have too many base learners. Trees can be too deep. If we think we're observing overfitting, we make our model architecture less complex.
- **Regularization:** Regularization refers to many sophisticated mathematical approaches for reducing the strength, complexity, or number of learned rules or parameters in an ML model. In fact, many types of ML models now incorporate multiple options for regularization, so make sure we employ these options to decrease the likelihood of overfitting.

- **Simpler Hypothesis Model Families:** Some ML models will be more complex than others out-of-the-box. If our neural network or gradient boosting machine (GBM) look to be overfit, we can try a less complex decision tree or linear model.

Overfitting is traditionally seen as the Achilles' heel of ML. While it is one the most likely bugs to encounter, it's also just one of many possible technical risks to consider from a safety and performance perspective. As with leakage, as ML systems become more complex, overfitting becomes harder to detect. Always keep an untouched holdout set with which to estimate real-world performance before deployment. More general controls like documentation of validation schemes, model monitoring, and A/B testing of models on live data also need to be applied to prevent overfitting.

Shortcut Learning

Shortcut learning occurs when a complex ML system is thought to be learning and making decisions about one subject, say anomalies in lung scans or job interview performance, but it's actually learned about some simpler related concept, such as machine identification numbers or Zoom video call backgrounds. Shortcut learning tends to arise from entangled concepts in training data, a lack of construct validity, and failure to adequately consider and document assumptions and limitations. Use explainable models and explainable AI (XAI) techniques to understand what learned mechanisms are driving model decisions, and we make sure we understand how our ML system makes scientifically valid decisions.

Underfitting

If someone tells us a statistic about a set of data, we might wonder how much data that statistic is based on, and whether that data was of high enough quality to be trustworthy. What if someone told us they had millions, billions, or even trillions of statistics for us to consider? They would need lots of data to make a case that all these statistics were meaningful. Just like averages and other statistics, each parameter or rule within an ML model is learned from data. Big ML models need lots of data to learn enough to make their millions, billions, or trillions of learned mechanisms meaningful. Underfitting happens when a complex ML algorithm doesn't have enough training data, constraints, or other input information, and it learns just a few generalizable concepts from training data, but not enough specifics to be useful when deployed. Underfitting can be diagnosed by poor performance on both training and validation data. Another piece of evidence for underfitting is if our model residuals have significantly more structure than random noise. This suggests that there are meaningful patterns in the data going undetected by our model, and it's another reason we examine our residuals for model debugging. We can mitigate underfitting by increasing the complexity of our models or preferably, providing more training data. We can also provide more input information in other ways, such as new features,

Bayesian priors applied to our model parameter distributions, or various types of architectural or optimization constraints.

Underspecification

Forty researchers recently published *Underspecification Presents Challenges for Credibility in Modern Machine Learning*. This paper gives a name to a problem that has existed for decades, *underspecification*. Underspecification arises from the core ML concept of the multiplicity of good models, sometimes also called the Rashomon effect. For any given data set, there are many accurate ML models. How many? Vastly more than human technicians have any chance of understanding in most cases. While we use validation data to select a good model from many models attempted during training, validation-based model selection is not a strong enough control to ensure we picked the best model - or even a servicable model - for deployment. Say that for some dataset there are a million total good ML models based on training data and the large number of potential hypothesis models. Selecting by validation data may cut that number of models down to a pool of one hundred total models. Even in this simple scenario, we'd still only have a 1 in 100 chance of picking the right model for deployment. How can we increase those odds? By injecting domain knowledge into ML models. By combining validation-based model selection with domain-informed constraints, we have a much better chance at selecting a viable model for the job at hand.

Happily, testing for underspecification can be fairly straightforward. One major symptom of underspecification is model performance that's dependent on computational hyperparameters that are not related to the structure of the domain, data, or model. If our model's performance varies due to random seeds, number of threads or GPUs, or other computational settings, our model is probably underspecified. Another test for underspecification is illustrated in Figure [Figure 3-1](#).

Error Metrics for PAY_0											
	Prevalence	Accuracy	True Positive Rate	Precision	Specificity	Negative Predicted Value	False Positive Rate	False Discovery Rate	False Negative Rate	False Omissions Rate	
PAY_0											
-2	0.124	0.864	0.099	0.333	0.972	0.884	0.028	0.667	0.901	0.116	
-1	0.168	0.816	0.206	0.406	0.939	0.854	0.061	0.594	0.794	0.146	
0	0.121	0.867	0.107	0.341	0.972	0.888	0.028	0.659	0.893	0.112	
1	0.325	0.491	0.903	0.292	0.972	0.862	0.708	0.619	0.097	0.138	
2	0.709	0.709	1	0.709	0	0.5	1	0.291	0	0.5	
3	0.748	0.748	1	0.748	0	0.5	1	0.252	0	0.5	
4	0.571	0.571	1	0.571	0	0.5	1	0.429	0	0.5	
5	0.444	0.444	1	0.444	0	0.5	1	0.556	0	0.5	
6	0.25	0.25	1	0.25	0	0.5	1	0.75	0	0.5	
7	0.5	0.5	1	0.5	0	0.5	1	0.5	0	0.5	
8	0.75	0.75	1	0.75	0	0.5	1	0.25	0	0.5	
Error Metrics for SEX											
	Prevalence	Accuracy	True Positive Rate	Precision	Specificity	Negative Predicted Value	False Positive Rate	False Discovery Rate	False Negative Rate	False Omissions Rate	
SEX											
Male	0.235	0.782	0.626	0.531	0.83	0.879	0.17	0.469	0.374	0.121	
Female	0.209	0.797	0.552	0.514	0.862	0.879	0.138	0.486	0.448	0.121	

Figure 3-1. Anayzing accuracy and errors across key segments is an important debugging method for detecting bias, underspecification, and other serious ML bugs.

Figure Figure 3-1 displays several error and accuracy measures across important segments in the example training data and model. Here a noticeable shift in performance for segments defined by higher values of the important feature PAY_0 points to a potential underspecification problem, likely due to data sparsity in that region of the training data. (Performance across segments defined by SEX is more equally balanced, which a good sign from a bias testing perspective, but certainly not the only test to be considered for bias problems.) Fixing underspecification tends to involve applying real-world knowledge to ML algorithms. Such domain-informed mechanisms include graph connections, monotonicity constraints, interaction constraints, beta constraints, or other architectural constraints.



Nearly all of the bugs discussed in this section, chapter, and book can affect certain segments of data more than others. For optimal performance it's important to test for weakspots (performance quality), over and underfitting, instability, distribution shifts and other issues across different kinds of segments in training, validation, test or holdout data.

Each of the ML bugs discussed in this subsection has real-world safety and performance ramifications. A unifying theme across these bugs is they cause systems to perform differently than expected when deployed *in vivo* and over time. Unexpected performance leads to unexpected failures and AI incidents. Using knowledge of potential bugs and bug detection methods discussed here to ensure estimates of vali-

dation and test performance are relevant to deployed performance will go a long way toward preventing real-world incidents. Now that we know what bugs we're looking for, in terms of software, traditional assessment and ML math, next we'll address how to find these bugs with residual analysis, sensitivity analysis, benchmark models and other testing and monitoring approaches.

Residual Analysis

Residual analysis is another type of traditional model assessment that can be highly effective for ML models and ML systems. At its most basic level, residual analysis means learning from mistakes. That's an important thing to do in life, as well as in organizational ML systems. Moreover, residual analysis is a tried and true model diagnostic technique. This subsection will use an example and three generally applicable residual analysis techniques to apply this established discipline to ML.

Note that some of the figures below contain the features `r_DEFAULT_NEXT_MONTH`. `r_DEFAULT_NEXT_MONTH` is a logloss residual value, or a numeric measure of how far off the prediction, `p_DEFAULT_NEXT_MONTH`, is from the known correct answer, `DEFAULT_NEXT_MONTH`. Readers may also see demographic features in the dataset, like `SEX`, that are used for bias testing. For the most part, Chapter [Chapter 3](#) treats the example credit lending problem as a general predictive modeling exercise, and does not consider applicable fair lending regulations. See [Chapter 4](#) and Chapter 10 for in-depth discussions relating to bias management in ML that also address some legal and regulatory concerns.

Analysis and Visualizations of Residuals

Plotting overall and segmented residuals and examining them for tell-tale patterns of different kinds of problems is a long-running model diagnostic technique. Residual analysis can be applied to ML algorithms to great benefit with a bit of creativity and elbow grease. Simply plotting residuals for an entire dataset can be helpful, especially to spot outlying rows causing very large numeric errors or to analyze overall trends in errors. However, breaking residual values and plots down by feature and level is likely to be more informative. Even if we have a lot features or features with many categorical levels we're not off the hook. Start with the most important features and their most common levels. Look for strong patterns in residuals that violate the assumptions of our model. Many types of residuals should be randomly distributed, indicating the model has learned all important information from the data, aside from irreducible noise. If we spot strong patterns or other anomalies in residuals that have been broken down by feature and level we first determine whether these errors arise from data, and if not, we can use XAI techniques to track down issues in our model. Residual analysis is considered standard practice for important linear regression models. ML models are arguably higher-risk and more failure prone, so they need even more residual analysis.

Modeling Residuals

Modeling residuals with interpretable models is another great way to learn more about the mistakes our ML system could make. In Figure [Figure 3-2](#) below, we've trained a single, shallow decision tree on the residuals from a more complex model associated with customers who missed a credit card payment.

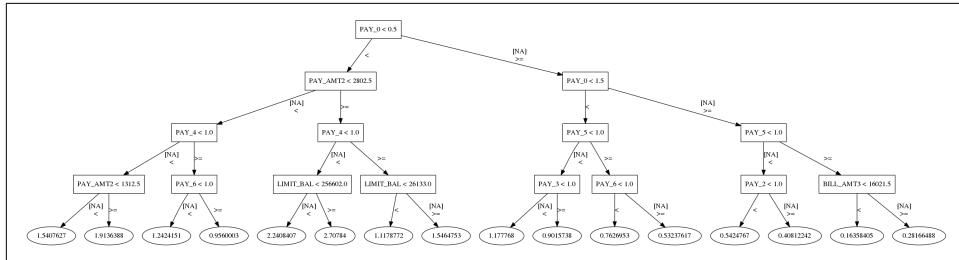


Figure 3-2. An interpretable decision tree model for customers who missed credit card payments. Adapted from [Responsible Machine Learning with Python](#).

This decision tree encodes rules that describe how the more complex model is wrong. For example, we can see the model generates the largest numeric residuals when someone misses a credit card payment, but looks like a great customer. When someone's most recent repayment status (PAY_0) is less than 0.5, when their second most recent payment amount (PAY_AMT2) is equal or above 2,802.50, their fourth most recent repayment status (PAY_4) is less than 1, and when their credit limit is greater than or equal to 256,602, we see logloss residuals of 2.71 on average. That's a big error rate that drags down our overall performance, and that can have bias ramifications if we make too many false negative guesses about already favored demographic groups.

Another intriguing use for the tree is to create model assertions, real-time business rules about model predictions, that could be used to flag when a wrong decision is occurring as it happens. In some cases, the assertions might simply alert model monitors that a wrong decision is likely being issued, or model assertions could involve corrective action, like routing this row of data to a more specialized model or to human case workers.

Local Contribution to Residuals

Plotting and modeling residuals are older techniques that are well-known to skilled practitioners. A more recent breakthrough has made it possible to calculate accurate Shapley value contributions to model errors. This means for any feature or row of any dataset, we can now know which features are driving model predictions, and which features are driving model errors. What this advance really means for ML is yet to be determined, but the possibilities are certainly intriguing. One obvious application for this new Shapley value technique is to compare feature importance for predictions to feature importance for residuals, as in Figure [Figure 3-3](#) below.

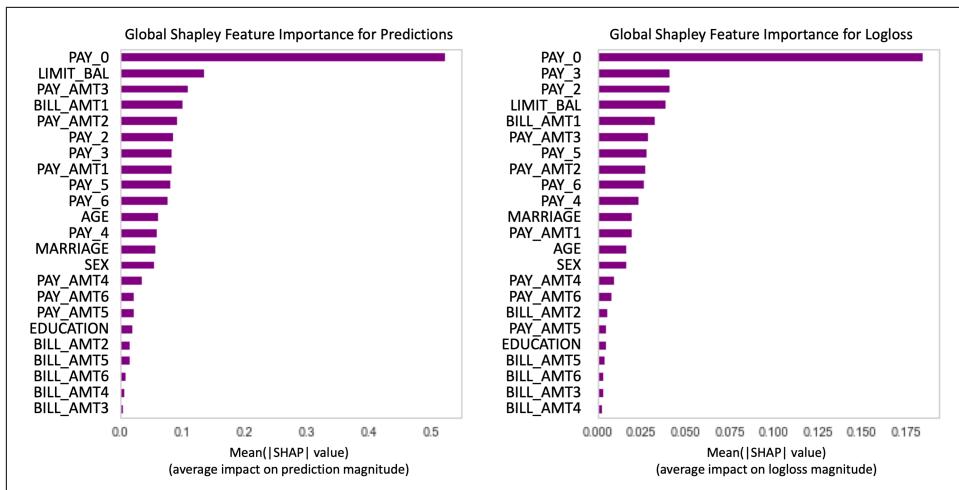


Figure 3-3. A comparison of Shapley feature importance to predictions and to model errors. Adapted from [Responsible Machine Learning with Python](#).

In Figure Figure 3-3, feature importance for predictions is shown at left, and feature importance to the errors of the model, as calculated by logloss, is shown on the right. We can see that PAY_0 dominates both predictions and errors, confirming that this model is too reliant on PAY_0 in general. We can also see that PAY_2 and PAY_3 are ranked higher for contributions to error than contributions to predictions. Given this, it might make sense to experiment with dropping, replacing or corrupting these features. Note that figure Figure 3-3 is made from aggregating Shapley contributions to logloss across an entire validation dataset. However, these quantities are calculated feature-by-feature and row-by-row. We could also apply this analysis across segments or demographic groups in our data, opening up interesting possibilities for detecting and remediating non-robust features for different subpopulations under the model.



Feature importance plots that look like Figure 3-3, with one feature drastically outweighing all the others, bode very poorly for *in vivo* reliability and security. If the distribution of that single important feature drifts our model performance is going to suffer. If hackers find a way to modify values of that feature, they can easily manipulate our predictions. When one feature dominates a model, we likely need a business rule relating to that feature instead of an ML model.

This ends our brief tour of residual analysis for ML. Of course there are other ways to study the errors of ML models. If readers prefer another way, then go for it! The important thing is to do some kind of residual analysis for all high-stakes ML sys-

tems. Along with sensitivity analysis, to be discussed in the next subsection, residual analysis is an essential tool in the ML model debugging kit.

Sensitivity Analysis

Unlike linear models, it's very hard to understand how ML models extrapolate or perform on new data, without testing them explicitly. That's the simple and powerful idea behind sensitivity analysis. Find or simulate data for interesting scenarios, then see how our model performs on that data. We really won't know how our ML system will perform in these scenarios unless we conduct basic sensitivity analysis. Of course, there are structured and more efficient variants of sensitivity analysis, such as in the [interpret](#) library from Microsoft Research. Another great option for sensitivity analysis, and a good place to start with more advanced model debugging techniques is random attacks, discussed in the [“Software Testing” on page 108](#) subsection. Many other approaches, like stress-testing, visualization, and adversarial example searches also provide standardized ways to conduct sensitivity analysis.

- **Stress-testing:** Stress-testing involves simulating data that represents realistic adverse scenarios, like recessions or pandemics, and making sure our ML models and any downstream business processes will hold up to the stress of the adverse situation.
- **Visualizations:** Visualizations like plots of accumulated local effect (ALE), individual conditional local effect (ICE), and partial dependence curves are a well-known and highly structured way to observe the performance of ML algorithms across various real or simulated values of input features. These plots can also reveal areas of data sparsity that can lead to weakspots in model performance.
- **Adversarial example searches:** Adversarial examples are rows of data that evoke surprising responses from ML models. Deep learning approaches can be used to generate adversarial examples for unstructured data, and ICE and genetic algorithms can be used to generate adversarial examples for structured data. Adversarial examples, and searching for them, are a great way to find local areas of instability in our ML response functions or decision boundaries that can cause incidents once deployed. As readers can see in Figure [Figure 3-4](#) below, an adversarial example search is a great way to put a model through its paces. See
- **Conformal approaches:** [Conformal approaches](#) that attempt to calculate empirical bounds for model predictions can help us understand model reliability through establishing the upper and lower limits of what can be expected from model outputs.
- **Perturbation tests:** Randomly perturbing validation, test, or holdout data to simulate different types of noise and drift, and then remeasuring ML model performance can also help establish the general bounds of model robustness. With this kind of perturbation testing we can understand and document the amount of

noise or shift that we know will break our model. One thing to keep in mind is that poor-performing rows often decline in performance faster than average rows under perturbation testing. Watch poor-performing rows carefully to understand if and when they drag the performance of the entire model down.

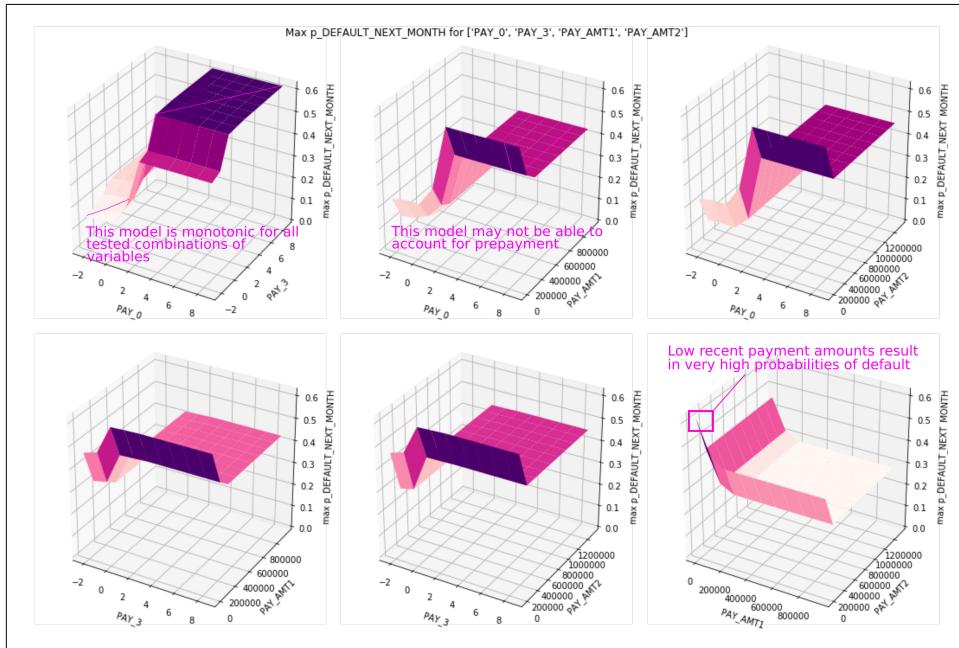


Figure 3-4. Results from an adversarial example search that reveal interesting model behavior. Adapted from [Responsible Machine Learning with Python](#).

Figure [Figure 3-4](#) was created by first finding an individual conditional expectation (ICE) curve that exhibited a large swing in predictions. Using the row of data responsible for that ICE curve as a seed, and perturbing the values of the four most important features in that row thousands of times and generating associated predictions lead to the numerous plots in Figure [Figure 3-4](#). The first finding of the adversarial example search is that this heuristic technique, based of ICE curves, enables us to generate adversarial examples that can evoke almost any response we want from the model. We found rows that reliably yield very low and very high predictions and everything in between. If this model was available via a prediction API, we could play it like a fiddle.

In the process of finding all those adversarial examples, we also learned things about our model. First, it is likely monotonic in general, and definitely monotonic across all the rows we simulated in the adversarial example search. Second, this model issues default predictions for people that make extremely high payments. Even if someone's most recent payment was one million dollars, and above their credit limit, this model

will issue default predictions once that person becomes two months late on their payments. This could pose problems for pre-payment. Do we really want to issue a default or delinquency decision for someone who prepaid millions of dollars but is now two months late on their most recent payments? Maybe, but it's likely not a decision that should be made quickly or automatically as this model would do. Third, it appears we may have found a route for a true adversarial example attack. Low most and second-most recent repayment amounts result in surprisingly sharp increases in probability of default. If a hacker wants to evoke high probability of default predictions from this model, setting PAY_AMT1 and PAY_AMT2 to low values could be how they do it.

Like we mentioned for residual analysis, readers may have other sensitivity analysis techniques in mind, and that's great. Just make sure you apply some form of realistic simulation testing to your ML models. The chapter case study is an example of the worst kind of outcome resulting from failing to conduct realistic simulation testing prior to deploying an ML system. This ends our brief discussion of sensitivity analysis. For those who would like to dive in even deeper, we recommend Chapter 19 of Kevin Murphy's free and open *Probabilistic Machine Learning: Advanced Topics*. Next, we'll discuss benchmark models in different contexts, another time-tested and commonsense model debugging approach.

Benchmark Models

Benchmark models have been discussed at numerous points within [Chapter 3](#). They are a very important safety and performance tool, with uses throughout the ML life-cycle. This subsection will discuss benchmark models in the context of model debugging and also summarize other critical uses.



When possible, compare ML model performance to a linear model or GLM performance benchmark. If the linear model beats the ML model, use the linear model.

The first way to use a benchmark model for debugging is to compare performance between a benchmark and the ML system in question. If the ML system does not outperform a simple benchmark, and many may not, it's back to the drawing board. Assuming a system passes this initial baseline test, benchmark models are a comparison tool used to interrogate mechanisms and find bugs within an ML system. For instance, data scientists can ask the question: "which predictions does my benchmark get right and my ML system get wrong?" Given that the benchmark should be well understood, it should be clear why it is correct, and this understanding should also provide some clues as to what the ML system is getting wrong. Benchmarks can also be used for reproducibility and model monitoring purposes as follows:

Reproducibility Benchmarks: Before making changes to a complex ML system it is imperative to have a reproducible benchmark from which to measure performance gains, or losses. A reproducible benchmark model is an ideal tool for such measurement tasks. If this model can be built into CI/CD processes that enable automated testing for reproducibility and comparison of new system changes to established benchmarks, even better.

Debugging Benchmarks: Comparing complex ML model mechanisms and predictions to a trusted, well-understood benchmark model's mechanisms and predictions is an effective way to spot ML bugs.

Monitoring Benchmarks: Comparing real-time predictions between a trusted benchmark model and a complex ML system is a way to catch serious ML bugs in real-time. If a trusted benchmark model and a complex ML system give noticeably different predictions for the same instance of new data, this can be a sign of an ML hack, data drift, or even bias and algorithmic discrimination. In such cases, benchmark predictions can be issued in place of ML system predictions, or predictions can be withheld until human analysts determine if the ML system prediction is valid.

If we set benchmarks up efficiently, it may even be possible to use the same model for all three tasks. A benchmark can be run before starting work to establish a baseline from which to improve performance, and that same model can be used in comparisons for debugging and model monitoring. When a new version of the system outperforms an older version in a reproducible manner, the ML model at the core of the system can become the new benchmark. If our organization can establish this kind of workflow, we'll be benchmarking and iterating our way to increased ML safety and performance.

Remediation: Fixing Bugs

The last step in debugging is fixing bugs. The previous subsections have outlined testing strategies, bugs to be on the lookout for, and a few specific fixes. This subsection outlines general ML bug-fixing approaches and discusses how they might be applied in the example debugging scenario. General strategies to consider during ML model debugging include:

- **Anomaly Detection:** Strange inputs and outputs are usually bad news for ML systems. These can be evidence of real-time security, bias, and safety and performance problem. Monitor ML system data queues and predictions for anomalies, record the occurrence of anomalies, and alert stakeholders to their presence when necessary.



A number of rule-based, statistical, and ML techniques can be used to detect anomalies in unseen data queues. These include data integrity constraints, confidence limits, control limits, autoencoders and isolation forests.

- **Experimental Design and Data Augmentation:** Collecting better data is often a fix-all for ML bugs. What's more, data collection doesn't have to be done in a trial-and-error fashion, nor do data scientists have to rely on data exhaust by products of other organizational processes for selecting training data. The mature science of design of experiment (DOE) has been used by data practitioners for decades to ensure they collect the right kind and amount of data for model training. Arrogance related to the perceived omnipotence of “big” data and overly compressed deployment time lines are the most common reasons data scientists don't practice DOE. Unfortunately, these are not scientific reasons to ignore DOE.
- **Model Assertions:** Model assertions are business rules applied to ML model predictions that correct for shortcomings in learned ML model logic. Using business rules to improve predictive models is a time-honored remediation technique that will likely be with us for decades to come. If there is a simple, logical rule that can be applied to correct a foreseeable ML model failure, don't be shy about implementing it. The best practitioners and organizations in the predictive analytics space have used this trick for decades.
- **Model Editing:** Given that ML models are software, those software artifacts can be edited to correct for any discovered bugs. Certain models, like GA2Ms or explainable boosting machines (EBMs) are designed to be edited for the purposes of model debugging. Other types of models may require more creativity to edit. Either way, editing must be justified by domain considerations, as it's likely to make performance on training data appear worse. ML models optimize toward lower error. If we edit this highly-optimized structure to make in-domain performance better, we'll likely worsen traditional assessment statistics. That's ok. We care more about *in vivo* safety, robustness and reliability than *in silica* test error.
- **Model Management and Monitoring:** ML models and the ML systems that house them are dynamic entities that must be monitored to the extent that resources allow. All mission-critical ML systems should be well-documented, inventoried, and monitored for security, bias, and safety and performance problems in real-time. When something starts to go wrong, stake-holders need to be alerted quickly. The next major section of Chapter [Chapter 3](#) gives more detailed treatment of model monitoring.
- **Monotonicity and Interaction Constraints:** Many ML bugs occur because ML models have too much flexibility and become untethered from reality due to

learning from biased and inaccurate training data. Constraining models with real-world knowledge is a general solution to several types of ML bugs. Monotonicity and interaction constraints, in popular tools like XGBoost, can help ML practitioners enforce logical domain assumptions in complex ML models.

- **Noise Injection and Strong Regularization:** Many ML algorithms come with options for regularization. However, if an ML model is over-emphasizing a certain feature, stronger or external regularization might need to be applied. L_0 regularization can be used to limit the number of rules or parameters in a model directly, and when necessary, manual noise injection can be used to corrupt signal from certain features to de-emphasize any undue importance in ML models.
- **The Scientific Method:** Confirmation bias between data scientists, ML engineers, their managers, and business partners often conspires to push half-baked demos out the door as products, based on the assumptions and limitations of *in silico* test data assessments. If we're able to follow the scientific method by recording a hypothesis about real-world results and objectively test that hypothesis with a designed experiment, we have much higher chances at *in vivo* success. See Chapter 12 for more thoughts on using the scientific method in ML.



Generally speaking ML is still more of an empirical science than an engineering discipline. We don't yet fully understand when ML works well and all the ways it can fail, especially when deployed *in vivo*. This means we have to apply the scientific method and avoid issues like confirmation bias to attain good real-world results. Simply using the right software and platforms, and following engineering best practices, does not mean our models will work well.

There's more detailed information regarding model debugging and the example data and model at the end of Chapter [Chapter 3](#). For now, we've learned quite a bit about model debugging and it's time to turn our attention to safety and performance for deployed ML systems.

Deployment

Once bugs are found and fixed, it's time to deploy our ML system to make real-world decisions. ML systems are much more dynamic than most traditional software systems. Even if system operators don't change any code or setting of the system, the results can still change. Once deployed, ML systems must be checked for in-domain safety and performance, they must be monitored, and their operators must be able to shut them off quickly. This last major subsection of Chapter [Chapter 3](#) will cover how to enhance safety and performance once an ML system is deployed: domain safety, model monitoring, and kill switches.

Domain Safety

Characteristics of Safe Machine Learning Systems

Some of the most important steps we can take to ensure an ML system interacts with the physical world in a safe way are:

- **Avoiding past failed designs:** ML systems that cause harm to humans or the environment should not be reimplemented, and their failures should be studied to improve safety conditions of future related systems.
- **Incident response plans:** Human operators should know what to do when a safety incident occurs.
- **In-domain testing:** Test data assessments, simulations, and debugging by data scientists are not enough to ensure safety. Systems should be tested in realistic *in vivo* conditions by domain and safety experts.
- **Kill switches:** Systems should be able to be shutoff quickly and remotely when monitoring reveals risky or dangerous conditions.
- **Manual prediction limits:** Limits on system behaviors should be set by operators where appropriate.
- **Real-time monitoring:** Humans should be alerted when a system enters a risky or dangerous state, and kill switches or redundant functionality should be enacted quickly (or automatically).
- **Redundancy:** Systems that perform safety- or mission-critical activities should have redundant functionality at the ready if incidents occur or monitoring indicates the system has entered a risky or dangerous state.

Domain safety means safety in the real world. This is very different than standard model assessment, or even enhanced model debugging. How can practitioners work toward real-world safety goals? A/B testing and champion challenger methodologies allow for some amount of testing in real-time operating environments. Process controls, like enumerating foreseeable incidents, implementing controls to address those potential incidents, and testing those controls under realistic or stressful conditions are also important for solid *in vivo* performance. To make up for incidents that can't be predicted, we apply chaos testing, random attacks, and manual prediction limits to our ML system outputs.

- **Foreseeable real-world incidents:** A/B testing and champion-challenger approaches, in which models are tested against one another on live data streams or under other realistic conditions are a first step toward robust in-domain testing. Beyond these somewhat standard practices, resources should be spent on domain experts and thinking through possible incidents. For example common

failure modes in credit lending include bias and algorithmic discrimination, lack of transparency, and poor performance during recessions. For other applications, say autonomous vehicles, there are numerous ways they could accidentally or intentionally cause harm. Once potential incidents are recorded, then safety controls can be adopted for the most likely or most serious potential incidents. In credit lending, models are tested for bias, explanations are provided to consumers via adverse action notices, and models are monitored to catch performance degradation quickly. In autonomous vehicles, we still have a lot to learn as the case study below will show. Regardless of the application, safety controls must be tested, and these tests should be realistic and performed in collaboration with domain experts. When it comes to human safety, simulations run by data scientists are not enough. Safety controls need to be tested and hardened *in vivo* and in coordination with people who have a deep understanding of safety in the application domain.

- **Unforeseeable real-world incidents:** Interactions between ML systems and their environments can be complex and surprising. For high-stakes ML systems, it's best to admit that unforeseeable incidents can occur. We can try to catch some of these potential surprises before they occur with chaos testing and random attacks. Important ML systems should be tested in strange and chaotic use cases and exposed to large amounts of random input data. While these are time- and resource-consuming tests, they are one of the few tools available to test for so-called "unknown unknowns." Given that no testing regime can catch every problem, it's also ideal to apply common sense prediction limits to systems. For instance, large loans or interest rates should not be issued without some kind of human oversight. Nor should autonomous vehicles be allowed to travel at very high speeds without human intervention. Some actions simply should not be performed purely automatically as of today and prediction limits are one way to implement that kind of control.

Another key aspect of domain safety is knowing if problems are occurring. Sometimes glitches can be caught before they grow into harmful incidents. To catch problems quickly, ML systems must be monitored. If incidents are detected, incident response plans or kill-switches may need to be activated.

Model Monitoring

It's been mentioned numerous times in Chapter [Chapter 3](#), but important ML systems must be monitored once deployed. This subsection focuses on the technical aspects of model monitoring. It outlines the basics of model decay, robustness, and concept drift bugs, how to detect and address drift, the importance of measuring multiple key performance indicators (KPIs) in monitoring, and it briefly highlights a few other notable model monitoring concepts.

Model Decay and Concept Drift

No matter what we call it, the data coming into an ML system is likely to drift away from the data on which the system was trained. The change in the distribution of input values over time is sometimes labeled “data drift.” The statistical properties of what we’re trying to predict can also drift, and sometimes this is known specifically as “concept drift.” The COVID-19 crisis is likely one of history’s best examples of these phenomena. At the height of the pandemic, there was likely a very strong drift toward more cautious consumer behavior accompanied by an overall change in late payment and credit default distributions. These kinds of shifts are painful to live through, and they can wreak havoc on an ML system’s accuracy. It’s important to note that we sometimes make our own concept drift issues, by using engaging in *off-label use* of ML models.



Both input data and predictions can drift. Both types of drift can be monitored, and the two types of drift may or may not be directly related. When performance degrades without significant input drift this may be due to real-world *concept drift*. We sometimes cause concept drift and degraded performance ourselves by engaging in *off-label use*, or using a model in scenarios for which it was not designed.

Detecting and Addressing Drift

The best approach to detect drift is to monitor the statistical properties of live data — both input variables and predictions. Once a mechanism has been put in place to monitor statistical properties, we can set alerts or alarms to notify stakeholders when there is a significant drift. Testing inputs is usually the easiest way to start detecting drift. This is because sometimes true data labels, i.e., true outcome values associated with ML system predictions, cannot be known for long periods of time. In contrast, input data values are available immediately whenever an ML system must generate a prediction or output. So, if current input data properties have changed from the training data properties, we likely have a problem on our hands. Watching ML system outputs for drift can be more difficult due to information needed to compare current and training quality being unavailable immediately. (Think about mortgage default versus online advertising — default doesn’t happen at the same pace as clicking on an online advertisement.) The basic idea for monitoring predictions is to watch predictions in real-time and look for drift and anomalies, potentially using methodologies such as statistical tests, control limits, and rules or ML algorithms to catch outliers. And when known outcomes become available, test for degradation in model performance and sustained bias management quickly and frequently.

There are known strategies to address inevitable drift and model decay. These include:

- Refreshing an ML system with extended training data containing some amount of new data.
- Refreshing or retraining ML systems frequently.
- Refreshing or retraining an ML system when drift is detected.

It should be noted that any type of retraining of ML models in production should be subject to the risk mitigation techniques discussed in Chapter [Chapter 3](#) and elsewhere in this book — just like they should be applied to the initial training of an ML system.

Monitoring Multiple Key Performance Indicators

Most discussions of model monitoring focus on model accuracy as the primary key performance indicator (KPI). Yet, bias, security vulnerabilities, and privacy harms can, and likely should, be monitored as well. The same bias testing that was done at training time can be applied when new known outcomes become available. Numerous other strategies, discussed elsewhere in [Chapter 5](#) and [Chapter 9](#), can be used to detect malicious activities that could compromise system security or privacy. Perhaps the most crucial KPI to measure, if at all possible, is the actual impact of the ML system. Whether it's saving or generating money, or saving lives, measuring the intended outcome and actual value of the ML system can lead to critical organizational insights. Assign monetary or other values to confusion matrix cells in classifications problems, and to residual units in regression problems, as a first step toward estimating actual business value. See Chapter 8 for a basic example of estimating business value.

Out-of-range Values

Training data can never cover all of the data an ML system might encounter once deployed. Most ML algorithms and prediction functions do not handle out-of-range data well, and may simply issue an average prediction or crash, and do so without notifying application software or system operators. ML system operators should make specific arrangements to handle data, such as large magnitude numeric values, rare categorical values, or missing values that were not encountered during training so that ML systems will operate normally and warn users when they encounter out-of-range data.

Anomaly Detection and Benchmark Models

Anomaly detection and benchmark models round out the technical discussion of model monitoring in this subsection. These topics have been treated elsewhere in Chapter [Chapter 3](#), and are touched on briefly here in the monitoring context.

- **Anomaly Detection:** Strange input or output values in an ML system can be indicative of stability problems or security and privacy vulnerabilities. It's possible to use statistics, ML, and business rules to monitor anomalous behavior in both inputs and outputs, and across an entire ML system. Record any such detected anomaly, report them to stakeholders, and be ready to take more drastic action when necessary.
- **Benchmark Models:** Comparing simpler benchmark models and ML system predictions as part of model monitoring can help to catch stability, fairness, or security anomalies in near real-time. A benchmark model should be more stable, easier to confirm as minimally discriminatory, and should be harder to hack. We use a highly transparent benchmark model and our more complex ML system together when scoring new data, then compare our ML system predictions against the trusted benchmark prediction in real-time. If the difference between the ML system and the benchmark is above some reasonable threshold, then fall back to issuing the benchmark model's prediction or send the row of data for more review.

Whether it's out-of-range values in new data, disappointing KPIs, drift, or anomalies — these real-time problems are where rubber meets road for AI incidents. If we're monitoring detects these issues, a natural inclination will be to turn the system off, and the next subsection addresses kill switches for ML systems.

Kill Switches

Kill switches are rarely single switches or scripts, but a set of business and technical processes bundled together that serve to turn an ML system off — to the degree that's possible. There's a lot to consider before flipping a proverbial kill switch. ML system outputs often feed into downstream business processes, sometimes including other ML systems. These systems and business processes can be mission critical, for example, an ML system used for credit underwriting or e-retail payment verification. To turn off an ML system, we'll not only need the right technical know-how and personnel available, but we also need an understanding of the system's place inside of broader organizational processes. During an ongoing AI incident is a bad time to start thinking about turning off a fatally flawed ML system. So, kill processes and kill switches are a great addition to our ML system documentation and AI incident response plans (see Chapter [Chapter 1](#)). This way, when the time comes to kill an ML system, our organization can be ready to make a quick and informed decision. Hopefully we'll never be in a position where flipping an ML system kill switch is necessary, but unfortunately AI incidents have grown more common in recent years. When technical remediation methods are applied along side cultural competencies and business processes for risk mitigation, safety and performance of ML systems is enhanced. When these controls are not applied, bad things can happen.

Case Study: Death by Autonomous Vehicle

On the night of March 18th 2018, Elaine Herzberg was walking a bicycle across a wide intersection in Tempe, Arizona. In what has become one of the most high-profile AI incidents, she was struck by an autonomous Uber test vehicle traveling at roughly 40 mph. According to the National Safety Transportation Board (NTSB), the test vehicle driver, who was obligated to take control of the vehicle in emergency situations, was distracted by a smart phone. The self driving ML system also failed to save Ms. Herzberg. The system did not identify Ms. Herzberg until 1.2 seconds before impact, too late to prevent a brutal crash.

Fallout

Autonomous vehicles are thought to offer safety benefits over today's status quo of human-operated vehicles. While self-driving cars have driven millions of miles with no fatalities, they've yet to deliver on the original promise of safer roads through ML driving. The NTSB's report [states](#) that this Uber's, "system design did not include a consideration for jaywalking pedestrians." The report also criticized lax risk assessments and immature safety culture at the company. Furthermore, an Uber employee raised serious concerns about 37 crashes in the previous 18 months and common problems with test vehicle drivers just days before the Tempe incident. As a result of the Tempe crash, Uber's autonomous vehicle testing was stopped in four other cities and governments around the US and Canada began re-examining safety protocols for self-driving vehicle tests. The driver has been charged with negligent homicide. Uber has been excused from criminal liability, but came to a monetary settlement with the deceased's family. The city of Tempe and State of Arizona were also sued by Ms. Herzberg's family for \$10 million each.

An Unprepared Legal System

It must be noted that the legal system in the US is somewhat unprepared for the reality of AI incidents, potentially leaving employees, consumers and the general public largely unprotected from the unique dangers of ML systems operating in our midst. The EU Parliament has put forward a liability regime for ML systems that would mostly prevent large technology companies from escaping their share of the consequences in future incidents. In the US, any plans for Federal AI product safety regulations are still in a preliminary phase. In the interim, individual cases of AI safety incidents will likely be decided by lower courts with little education and experience in handling AI incidents, enabling Big Tech and other ML system operators to bring vastly asymmetric legal resources to bear against individuals caught up in incidents related to complex ML systems. Even for the companies and ML system operators, this legal limbo is not ideal. While the lack of regulation seems to benefit those with the most resources and expertise, it makes risk management and predicting the out-

comes of AI incidents more difficult. Regardless, future generations may judge us harshly for allowing the criminal liability of one of the first AI incidents, involving many data scientists and other highly paid professionals and executives, to be pinned solely on a safety driver of a supposedly automated vehicle.

Lessons Learned

What lessons learned from this and previous chapters could be applied to this case?

- **Lesson 1: Culture is important.** A mature safety culture is a broad risk control, bringing safety to the forefront of design and implementation work, and picking up the slack in corner cases that processes and technology miss. Learned from the last generation of life-changing commercial technologies, like aerospace and nuclear power, a more mature safety culture at Uber could have prevented this incident, especially since an employee raised serious concerns in the days before the crash.
- **Lesson 2: Mitigate foreseeable failure modes.** The NTSB concluded that Uber's software did not specifically consider jaywalking pedestrians as a failure mode. For anyone who's driven a car with pedestrians around, this should have been an easily foreseeable problem for which any self-driving car should be prepared. ML systems generally are not prepared for incidents unless their human engineers make them prepared. This incident shows us what happens when those preparations are not made in advance.
- **Lesson 3: Test ML systems in their operating domain.** After the crash, Uber stopped and reset its self-driving car program. After improvements, they were able to show via simulation that their new software would have started breaking 4 seconds before impact. Why wasn't the easily foreseeable reality of jaywalking pedestrians tested with these same in-domain simulations before the March 2018 crash? The public may never know. But enumerating failure modes and testing them in realistic scenarios could prevent our organization from having to answer these kinds of unpleasant questions.

A potential bonus lesson here is to consider not only accidental failures, like the Uber crash, but also malicious hacks against ML systems and the abuse of ML systems to commit violence. Terrorists have turned motor vehicles into deadly weapons before, so this is a known failure mode. Precautions must be taken in autonomous vehicles, and in driving assistance features, to prevent hacking and violent outcomes. Regardless of whether it's an accident or a malicious attack, AI incidents will certainly kill more people. Our hope is that governments and other organizations will take ML safety seriously, and minimize the number of these somber incidents in the future.

Resources

- Talks, workshops, and educational materials related to model debugging and ML safety and performance:
 - *A Comprehensive Study on Deep Learning Bug Characteristics*
 - Debugging Machine Learning Models
 - Safe and Reliable Machine Learning
 - Testing and Debugging in Machine Learning.
 - *DQI: Measuring Data Quality in NLP*
 - *Identifying and Overcoming Common Data Mining Mistakes*
- Further reading and code examples related to model debugging examples:
 - Further reading: *Real-World Strategies for Model Debugging*
 - Code examples for Chapter 3:
 - https://nbviewer.jupyter.org/github/jphall663/interpretable_machine_learning_with_python/blob/master/resid_sens_analysis.ipynb
 - https://nbviewer.jupyter.org/github/jphall663/interpretable_machine_learning_with_python/blob/master/debugging_sens_analysis_redux.ipynb
 - https://nbviewer.jupyter.org/github/jphall663/interpretable_machine_learning_with_python/blob/master/debugging_resid_analysis_redux.ipynb

Managing Bias in Machine Learning

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Managing the harmful effects of bias in machine learning (ML) systems is about so much more than data, code and models. Our model’s average performance quality — the main way data scientists are taught to evaluate the goodness of a model — has little to do with whether it’s causing real-world bias harms. A perfectly accurate model can cause bias harms. Worse, all ML systems exhibit some level of bias, bias incidents appear to be some of the most common AI incidents (see Figure [Figure 4-1](#) below), bias in business processes often entails legal liability, and bias in ML models hurts people in the real world.

This chapter will put forward approaches for detecting and mitigating bias in a socio-technical fashion, at least to the best of our ability as practicing technicians. That means we’ll try to understand how ML system bias exists in its broader societal context. Why? *All* ML systems are sociotechnical. We know this might be hard to believe at first, so let’s think through one example. Let’s consider a model used to predict sensor failure for some internet of things (IoT) application, using only information from

other automated sensors. That model would likely have been trained by humans, or a human decided that model was needed. Moreover, the results from that model could be used to inform the ordering of new sensors, which could affect the employment of those at the manufacturing plant or those who repair or replace failing sensors. Finally, if our preventative maintenance model fails, people could likely be harmed. For every example we could think of that seemed purely technical, it became obvious that decision-making technologies like ML don't exist without interacting with humans in some way. (See [Figure 4-1](#).)

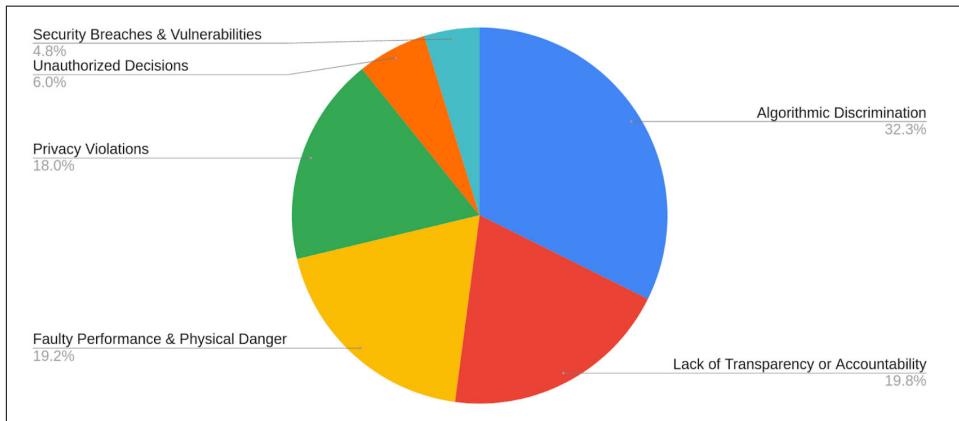


Figure 4-1. The frequency of different types of AI incidents based on a qualitative analysis of 169 publicly reported incidents between 1988 and February 1, 2021. Figure courtesy of BNH.AI.

This means there's no purely technical solution to bias in ML systems. If you want to jump right into the code for bias testing and bias remediation, see Chapter 10. However, we don't recommend this. You'll miss a lot of important information about what bias is and how to think about it in productive ways. This chapter starts out by defining bias using several different authoritative sources, and how to recognize our own cognitive biases that may affect the ML systems we build or the results our users interpret. The chapter then provides a broad overview of who tends to be harmed in AI bias incidents and what kinds of harms they experience. From there, we'll cover methods to test for bias in ML systems and discuss mitigating bias, both technical and sociotechnical approaches. Finally, the chapter will close with a case discussion of the Twitter image cropping algorithm.



NIST does not review, approve, condone or otherwise address any content in this book, including any claims relating to the AI RMF. All AI RMF content is simply the authors' opinions and in no way reflects an official position of NIST or any official or unofficial relationship between NIST and the book or the authors.

NIST AI RMF Crosswalk

Chapter Section NIST AI RMF Sub-categories

Applicable AI trustworthiness characteristics include: Managed_Bias, Transparent_and_Accountable, Valid_and_Reliable

See also:

*

ISO and NIST Definitions for Bias

The International Organization for Standardization (ISO) defines bias as “the degree to which a reference value deviates from the truth” in [Statistics — Vocabulary and symbols — Part 1](#). This is a very general notion of bias, but bias is a complex and heterogeneous phenomenon. Yet, in all of its instances, it’s about some systematic deviation from the truth. In decision-making tasks, bias takes on many forms. It’s wrong to deny people employment due to the level of melanin in their skin. It’s wrong to think an idea is correct just because it’s the first thing that comes to mind, and it’s wrong to train an ML model on incomplete and unrepresentative data. In recent work from the U.S. National Institute for Standards and Technology (NIST), [Towards a Standard for Identifying and Managing Bias in Artificial Intelligence \(SP1270\)](#), the subject of bias is divided into three major categories that align these examples of bias: systemic, human, and statistical biases.

Systemic Bias

Often when we say bias in ML, we mean systemic biases. These are historical, social, and institutional biases that are, sadly, so baked-in into our lives that they show up in ML training data and design choices by default. A common consequence of systemic bias in ML models is the incorporation of demographic information into system mechanisms. This incorporation may be overt and explicit, such as large language models (LLMs) [repurposed to generate harmful and offensive content](#) that targets certain demographic groups. However, in practice, incorporation of demographic information into decision-making processes tends to be unintentional and implicit, leading to differential outcomes rates or outcome prevalence across demographic groups, for example, by matching more men’s resumes to higher paying job descriptions, or design problems that exclude certain groups of users (e.g., those with physical disabilities) from interacting with a system.

Statistical Bias

Statistical biases can be thought of as mistakes made by humans in the specification of ML systems, or emergent phenomena like concept drift, that affect ML models and are difficult for humans to mitigate. Other common types of statistical biases include predictions based on unrepresentative training data, or error propagation and feedback loops. One potential indicator of statistical bias in ML models is differential performance quality across demographic groups. Differential validity for an ML model is a particular type of bias, somewhat distinct from the differing outcome rates or outcome prevalence described above for human biases. Moreover, there is a **documented tension** between maximizing ML performance and positive outcomes across demographic groups. Statistical biases may also lead to serious AI incidents, for example when concept drift in new data renders a system's decisions more wrong than right, or when feedback loops or error propagation lead to increasingly large volumes of bad predictions over a short timespan.

Human Biases and Data Science Culture

There are a number of human or cognitive biases that can come into play with both the individuals and the teams that design, implement, and maintain ML systems. NIST SP1270 discusses more than we'll take on here, and it's definitely time well spent to understand Figure 2 and the glossary in the NSIT 1270 guidance paper. Below we'll highlight the human biases that we've seen most frequently affecting both data scientists and users of ML systems.

- **Anchoring:** When a particular reference point, or *anchor*, has an undue effect on people's decisions. This is like when a benchmark for a state of the art deep learning model is stuck at 0.4 AUC for a long time, and someone comes along and scores 0.403 AUC. We shouldn't think that's important, but we're anchored to 0.4.
- **Availability Hueristic:** People tend to overweight what comes easily or quickly to mind in decision-making processes. Said another way, we often confuse *easy to remember* with *correct*.
- **Confirmation Bias:** A cognitive bias where people tend to prefer information that aligns with, or confirms, their existing beliefs. Confirmation bias is a big problem in ML systems when we trick ourselves into thinking our ML models work better than they actually do.
- **Dunning-Kruger effect:** The tendency of people with low ability in a given area or task to overestimate their self-assessed ability. This happens when we allow ourselves to think we're experts at something because we can import `sklearn` and run `model.fit()`, but we're actually not.
- **Funding bias:** A bias toward highlighting or promoting results that support or satisfy the funding agency or financial supporter of a project. We do what makes

our bosses happy, what makes our investors happy, and what increases our own salaries. Real science needs safeguards that prevent its progress from being altered by biased financial interests.

- **Groupthink:** When people in a group tend to make nonoptimal decisions based on their desire to conform to the group or fear dissenting with the group. It's hard to disagree with your team, even when you're right.
- **McNamara Fallacy:** The pervasive belief that quantitative information and automation are the best ways to make decisions. The underlying belief that data-driven decisions are more accurate and objective than human, or other types of decisions, is tainted by a known bias.
- **Techno-chauvanism:** The belief that technology is always the solution.

Groups of data scientists may often fall victim to a mixture of confirmation bias, the Dunning-Kruger effect, funding bias, groupthink, and the McNamara fallacy. How many times have we trained a model and then fudged something about the model or data to get better test data results? That sounds a lot like confirmation bias. We'll dig into this major problem later in Chapter 12. Remember that time you read a Medium post and turned around and wrote a Medium tutorial on the same subject? That could have been the Dunning-Kruger effect. Remember that big project your company just knew would work and invested in heavily, but your team knew it wouldn't work and still rushed it over the finish line? That could be funding bias or groupthink. Remember that time your team really wanted to use PyTorch and GPUs but you knew it wouldn't help with the problem at hand and you didn't want to keep the team from getting new shiny hardware? That's probably groupthink and funding bias again. Remember how we all got into this new career that promised fast, standardized, accurate, and objective decision-making for the information economy? We all need to think harder about the McNamara fallacy and double check some of those fundamental assumptions. On the user side, cognitive biases, like the availability heuristic or anchoring, can also influence how users interpret system outcomes. All these biases can lead to inappropriate and overly optimistic design choices, in turn leading to poor performance when a system is deployed, and finally, leading to harms for system users or operators.

We'll get into the harms that can arise and what to do about these problems below. For now we want to highlight a common sense mitigant that is also a theme of this chapter. You cannot treat bias properly without looking at a problem from many different perspectives. Step 0 of fighting bias in ML is having the right people in the room when decisions about the system are made. This doesn't mean just a larger group of white or Asian men with technical degrees and technical work experience. To avoid the blind spots that allow biased ML models to cause harm, you'll need many different types of perspectives informing system design, implementation, and maintenance decisions. Yes, different demographic perspectives, including from those

with disabilities, but also different educational backgrounds, like social scientists or attorneys, and you'll need people with domain expertise.

Also, consider the *digital divide*. A shocking percentage of the population still doesn't have access to good internet, new computers, and information like this book. If we're drawing conclusions about our users, we need to remember there is a solid chunk of the population that's not going to be included in user statistics. Leaving potential users out is a huge source of bias and harm in system design, bias-testing, and other crucial junctures in the ML lifecycle. Success in ML today still requires people who have a keen understanding of the real-world problem we're trying to solve, and what potential users might be left out of our design, data and testing.

United States Legal Notions of ML Bias

We should be aware of the many important legal notions of bias. However, it's also important to understand that law is extremely complex and context-sensitive. Just knowing a few definitions is still light years away from having any real expertise on these matters. As data scientists, legal matters are an area where we should not let the Dunning-Kruger effect takeover. With those caveats, let's dive into a basic overview.



Now is the time to reach out to your legal team if you have any questions or concerns about bias in ML models. Dealing with bias in ML models is one of the most difficult and serious issues in the information economy. Data scientists need help from lawyers to properly address bias risks.

In the U.S., bias in decision-making processes that affect the public has been regulated for decades. A major focus of early laws and regulations in the U.S. was employment matters. Notions like protected groups, disparate treatment and disparate impact (defined below) have now spread to a broader set of laws in consumer finance and housing, and are even being cited in brand new local laws today, like the New York City audit requirement for artificial intelligence (AI) used in hiring. Non-discrimination in the E.U. is addressed in the Charter of Fundamental Rights, the European Convention on Human Rights, and in the Treaty on the Functioning of the EU, and crucially for us, in aspects of the proposed EU AI Act. While it's impossible to summarize these laws and regulations, even on the U.S. side, the definitions below are what we think are most directly applicable to a data scientist's daily work. They are drawn, very roughly, from laws like the Civil Rights Act, Fair Housing Act (FHA), Equal Employment Opportunity Commission (EEOC) regulations, the Equal Credit Opportunity Act (ECOA), and the Americans with Disabilities Act (ADA). The definitions we highlight below cover legal ideas about what traits are protected under law and what they are protected against.

- **Protected Groups:** In the US, many laws and regulations prohibit discrimination based race, sex, age, religious affiliation, national origin, and disability status, among others. Prohibited bases under FHA include race, color, religion, national origin, sex, familial status, and disability. The GDPR, as an example of one non-US regulation, prohibits the use of personal data about racial or ethnic origin, political opinions, and other categories somewhat analogous to US protected groups. One implication of protected groups is that we often use white people and males as the control groups for bias testing.
- **Disparate Treatment:** Disparate treatment is one type of legally problematic discrimination. It's a decision that treats an individual less favorably than similarly situated individuals because of a protected characteristic such as race, sex, or other trait. For data scientists working on employment, housing or credit applications, this means you should probably be very careful when using demographic data in ML models, and even in your bias remediation techniques. Once demographic data is used as input in a model, that could mean that a decision for someone could be different just because of their demographics, and that could be disparate treatment in some cases.



Concerns about disparate treatment, or more general social bias, are why we typically try not to use demographic markers as direct inputs to ML models. To be conservative, demographic markers should *not* be used as model inputs in most common scenarios, but should be used for bias testing or monitoring purposes.

- **Disparate Impact:** Disparate impact is another kind of legally concerning discrimination. It's basically about different *outcome* rates or prevalence across demographic groups. Disparate impact is more formally defined as a facially neutral policy or practice that disproportionately harms a group based on a protected trait. For data scientists, disparate impact tends to happen when we don't use demographic data as inputs, but we use something correlated to demographic data as an input. Consider credit scores. They are fairly accurate predictor of default, so they are often seen as valid to use in predictive models in credit scoring. However, they are correlated to race, where some minority groups have lower credit scores on average. If you use credit score in a model, this tends to result in certain minority groups having lower proportions of positive outcomes, and that's a common example of disparate impact. (That's also why several states have started to restrict the use of credit scores in some insurance-related decisions.)
- **Differential Validity:** Differential validity is a construct that comes up sometimes in employment. Where disparate impact is often about different outcome

rates across demographic groups, differential validity is more about different *performance quality* across groups. It happens when an employment test is a better indicator of job performance for some groups than for others. Differential validity is important because the mathematical underpinning, not the legal construct, generalizes to nearly all ML models. It's common to use unrepresentative training data and to build a model that performs better for some groups than for others, and a lot of more recent bias testing approaches focus on this type of bias.

- **Screenout:** Screenout is a very important type of discrimination that highlights the socio-technical nature of ML systems and proves that testing and balancing the scores of a model is simply insufficient to protect against bias. Screen out happens when a person with a disability, such as limited vision or difficulties with fine motor skills, is unable to interact with some employment assessment, and is screened out of a job or promotion by default. Screen out is a serious issue, and the EEOC and Department of Labor are [paying attention](#) to the use of ML in this space. Note that screen out cannot necessarily be fixed by mathematical bias testing or bias remediation, it typically must be addressed in the design phase of the system, where designers ensure those with disabilities are able to work with the end-products interfaces. Screenout also highlights why you might want perspectives from lawyers or those with disabilities when building ML systems. Without those perspectives, it's all too easy to forget about people with disabilities when building ML systems, and that can give rise to legal liabilities in some cases.

This concludes our discussion on general definitions of bias. As you can see, it's a complex and multifaceted topic with all kinds of human, scientific and legal concerns coming into play. We'll add to these definitions with more specific, but probably more fraught, mathematical definitions of bias when we discuss bias-testing later in the chapter. Next we'll outline who tends to experience bias and related harms from ML systems.

Who Tends to Experience Bias from ML Systems

Any demographic group can experience bias and related harms when interacting with an ML system, but history tells us certain groups are more likely to experience bias and harms more often. In fact, it's the nature of supervised learning — which only learns and repeats patterns from past recorded data, that tends to result in older people, those with disabilities, immigrants, people of color, women, and gender nonconforming individuals facing more bias from ML systems. Said another way, those who experience discrimination in the real-world, or in the digital world, will likely also experience it when dealing with ML systems because all that discrimination is recorded in data and used to train ML models. The groups below are often protected groups under various laws, but not always. They will often, but not always, be the comparison group in bias testing for statistical parity of scores or outcomes between

two demographic groups. Keep in mind we also have to remember intersectional groups, like Hispanic women, who are members of more than one group.

Many people belong to multiple protected or marginalized groups. The important concept of intersectionality tells us that societal harm is concentrated among those who occupy multiple protected groups and that bias **should not only be analyzed as affecting marginalized groups along single group dimensions**. For example, AI ethics researchers **recently showed** that some commercially available facial recognition systems have substantial gender classification accuracy disparities, with darker-skinned women being the most misclassified group. Finally, before defining these groups, it's also important to think of the McNamara Fallacy. Is it even right to put nuanced human beings into this kind of blunted taxonomy? Probably not, and it's likely that assigning to these simplistic groups, because it's easy to represent as a binary marker column in a database, is also a source of bias and potential harms. There's always a lot of caveats in managing bias in ML systems, so with those in mind, we tread carefully into defining simplistic demographic groups that tend to face more discrimination and are often used as the comparison groups in traditional bias testing.

- **Age:** Older people, typically those 40 and above, are more likely to experience discrimination in online content. The age cutoff could be older in more traditional applications like employment, housing, or consumer finance. However, participation in Medicare or the accumulation of financial wealth over a lifetime may make older people the favored group in other scenarios.
- **Disability:** Those with physical, mental, or emotional disabilities are perhaps some of the likeliest people to experience bias from ML systems. The idea, but likely not the legal construct, of screenout generalizes outside of employment. People with disabilities are often forgotten about during the design of ML systems, and no amount of mathematical bias testing or remediation can make up for that.
- **Immigration Status or National Origin:** People who live in a country in which they were not born, with any immigration status — including naturalized citizens, are known to face significant bias challenges.
- **Language:** Especially in online content, an important domain for ML systems, those who use languages other than English or who write in non-Latin scripts may be more likely to experience bias from ML systems.
- **Race and Ethnicity:** Races and ethnicities other than White people, including those who identify as more than one race, are commonly subject to bias and harm when interacting with ML systems. Some also prefer skin tone scales over traditional race or ethnicity labels, especially for computer vision tasks. The **Fitzpatrick** scale is an example of a skin tone scale.

- **Sex and Gender:** Sexes and genders other than cisgender men are more likely to experience bias and harms at hands of an ML system. And in online content, women are often favored but in harmful ways. Known as the *male gaze* phenomenon, media about women may be appealing and receive positive treatment, specifically because that content is oriented toward objectification, subjugation, and/or sexualization of women. This means in bias testing, women being favored in online content may not always be a good thing!
- **Intersectional Groups:** People who are in two or more of the groups above may experience bias or harms that are greater than the simple sum of the two broader groups to which they belong. All the bias testing and mitigation steps below should address intersectional groups.

Of course these are not the only groups of people who may experience bias from an ML model, and grouping people can be problematic no matter what the motivation. However, it's important to know where to start looking for bias, and we hope our list is sufficient for that purpose. Now that we know where to start looking for ML bias, let's discuss the most common harms that we should be mindful of.

Harms That People Experience

Many common types of harm occur in online or digital content. They occur frequently, too. Perhaps so frequently we may become blind to them. The list below highlights common harms and provides examples so that we can recognize them better when we see them next. These harms align closely with those laid out in Abagayle Lee Blank's *Computer vision machine learning and future-oriented ethics*, which describes cases in which these harms occur in computer vision.

- **Denigration:** Content that is actively derogatory or offensive (e.g., offensive content generated by chatbots like [Tay](#) or [Lee Luda](#)).
- **Erasure:** Erasure of content challenging dominant social paradigms or past harms suffered by marginalized groups (e.g., [suppressing content](#) that discusses racism or calls out white supremacy).
- **Ex-nomination:** Treating notions like whiteness, maleness or heterosexuality as central human norms (e.g., [online searches](#) returning a Barbie Doll as the first female result for “CEO”).
- **Mis-recognition:** Mistaking a person’s identity or failing to recognize someone’s humanity (e.g., [mis-recognizing Black people](#) in automated image tagging.).
- **Stereotyping:** The tendency to assign characteristics to all members of a group (e.g., large language models automatically associating [Muslims with violence](#)).

- **Underrepresentation:** The lack of representation of a sensitive attribute within a dataset category (e.g., using less training data associated with marginalized demographic groups in **facial recognition** systems).

Sometimes these harms may only cause effects limited to online or digital spaces, but as our digital lives begin to overlap more substantially with other parts of our lives, harms also spill over into the real world. Also, ML systems in healthcare, employment, education, or other high-risk areas can cause harm directly, by wrongfully denying people access to needed resources. Below we'll cover the most obvious types of real-world harms caused by ML systems: economic, physical, psychological, and reputational.

- **Economic Harms:** When an ML system reduces the economic opportunity or value of some activity (e.g., when men **see more ads** for better jobs than women).
- **Physical Harms:** When an ML system hurts or kills someone (e.g., when people **over-rely on self-driving automation**).
- **Psychological Harms:** When an ML system causes mental or emotional distress (e.g., when **disturbing content** is recommended to children).
- **Reputational Harms:** When an ML system diminishes the reputation of an individual or organization (e.g., a consumer credit product rollout is marred by **accusations** of discrimination).

Unfortunately, users or subjects of ML systems may experience additional harms or combinations of harms that manifest in strange ways. Before we get too deep in the weeds with different kinds of tests and thresholds in the next section, remember that checking in with your users to make sure they are not experiencing the harms discussed here, or other types of harms, is perhaps one of the most direct ways to track bias in ML systems. In fact, in the most basic sense, it matters much more whether people are experiencing harm than whether some set of scores passes a necessarily-flawed mathematical test. Think about these harms when designing your system, talk to your users to ensure they don't experience harm, and seek to mitigate harms.

Testing for Bias

If there's a chance some ML system could harm people, it should be tested for bias. The goal of this section is to cover the most common approaches for testing ML models for bias so you can get started with this important risk management task. Testing is neither straightforward nor conclusive. Just like in performance testing, a system can look fine on test data, and go on to fail or cause harm once deployed. Or a system could exhibit minimal bias at testing and deployment time, but drift into making biased or harmful predictions over time. Moreover, there are many tests and effect size measurements, with known flaws and that conflict with one another. For a

good overview of these issues, see the YouTube Video of the [21 Definitions of Fairness and Their Politics](#) Fairness, Accountability, and Transparency in ML (FAccTML) conference talk by Princeton Professor Arvind Narayanan. For an in-depth mathematical analysis of why we can't simply minimize all bias metrics at once, checkout [Inherent Trade-Offs in the Fair Determination of Risk Scores](#). With these cautions in mind, let's start our tour of contemporary bias-testing approaches.

Testing Data

This section covers what's needed in training data to test for bias, and how to test that data for bias even before a model is trained. ML models learn from data. But no data is perfect or without bias. If systemic bias is represented in training data, that bias will likely manifest in the model's outputs. It's logical to start testing for bias in training data. But to do that, we have to assume some columns of data are available. At minimum, for each row of data, we need demographic markers, known outcomes (y , dependent variable, target feature, etc.) and later, we'll need model outcomes — predictions for regression models, and decisions and confidence scores or posterior probabilities for classification models. While there are a handful of testing approaches that don't require demographic markers, most accepted approaches require this data. Don't have it? Testing is going to be much more difficult, but we'll provide some guidance on inferring demographic marker labels too.

The need to know or infer demographic markers is a good example of why handling bias in ML requires holistic design thinking, and not just slapping another Python package onto the end of your pipeline. Demographic markers and individual-level data are also more sensitive from a privacy standpoint, and sometimes organizations don't collect this information for data privacy reasons. While the interplay of data privacy and nondiscrimination law is very complex, it's probably not the case that data privacy obligations override nondiscrimination obligations. But as data scientists, we can't answer such questions on our own. Any perceived conflict between data privacy and nondiscrimination requirements has to be addressed by attorneys and compliance specialists. Such complex legal considerations are an example of why addressing bias in ML necessitates engagement of a broad set of stakeholders.



In employment, consumer finance, or other areas where disparate treatment is prohibited, we need to check with our legal colleagues before changing our data based directly on protected class membership information, even if our intention is to mitigate bias.

By now you're probably starting to realize how challenging and complex bias-testing can be, and as technicians dealing with this complexity is not our sole responsibility. But we need to be aware of it and work within a broader team to address bias in ML systems. Let's transition to the job a technician when it comes to preparing data and

testing data for bias. If we have the data we need, we tend to look for three major issues: representativeness, distribution of outcomes, and proxies.

- **Representativeness:** The basic check to run here is to calculate the proportion of rows for each demographic group in the training data, with the idea that a model will struggle to learn about groups with only a small number of training data rows. Generally, proportions of different demographic groups in training data should reflect the population on which the model will be deployed. If it doesn't, we should probably collect more representative data. It's also possible to resample or reweigh a dataset to achieve better representativeness. However, if we're working in employment, consumer finance, or other areas where disparate treatment is prohibited, we really need to check with our legal colleagues before changing our data based directly on protected class membership information. If you're running into differential validity problems (described below) then rebalancing your training data to have larger or equal representation across groups may be in order. Balance among different classes may increase prediction quality across groups, but it may not help with, or may even worsen, imbalanced distributions of positive outcomes.
- **Distribution of Outcomes:** We need to know how outcomes (y variable values) are distributed across demographic groups because if the model learns some groups receive more positive outcomes than others, that can lead to disparate impact. We need to calculate a bi-variate distribution of y across each demographic group. If we see an imbalance of outcomes across groups then we can try to resample or reweigh our data training data, with certain legal caveats. More likely, we'll simply end up knowing that bias risks are serious for this model, and when we test its outcomes, we'll need to pay special attention and likely plan on some type of remediation.
- **Proxies:** In most business applications of ML, we should *not* be training models on demographic markers. (Those are for testing!) But even if we don't use demographic markers directly, information like name, address, credit score, educational details or facial images may encode a great deal of demographic information. Other types of information may proxy for demographic markers too. One way to find proxies is to build an adversarial model based on each input column and see if those models can predict any demographic marker. If they can predict a demographic marker then those columns encode demographic information and are likely demographic proxies. If possible, such proxies should be removed from training data. Proxies may also be more hidden, or latent, in training data. There's no standard technique to test for latent proxies, but you can apply the same adversarial modeling technique as described for direct proxies, except instead of using the features themselves you can use engineered interactions of features that you suspect may be serving as proxies. We also suggest having dedicated legal or compliance stakeholders vet each and every input feature

in your model with an eye towards proxy discrimination risk. If proxies cannot be removed or you suspect the presence of latent proxies, pay careful attention to bias testing results for system outcomes, and be prepared to take remediation steps later in the bias mitigation process.

The outlined tests and checks for representativeness, distribution of outcomes, and proxies in training data all rely on the presence of demographic group markers. So will most of the tests for model outcomes. If we don't have those demographic labels, then one accepted approach is to infer them. The [Bayesian improved surname geocoding \(BISG\)](#) approach infers race and ethnicity from name and postal code data. It's sad but true that U.S. society is still so segregated that ZIP code and name can predict race and ethnicity, often above 90% accuracy. This approach was developed by the Consumer Financial Protection Bureau (CFPB) and has a high level of credibility for bias testing in consumer finance. The CFPB even has code on their [GitHub](#) for BISG! If necessary, similar approaches may be used to [infer gender](#) from name, social security, or birth year.

Traditional Approaches: Testing for Equivalent Outcomes

Once we've assessed our data for bias, made sure we have the information needed to perform bias testing, and trained a model, it's time to test its outcomes for bias. (Yes, this is in addition to model debugging and red-teaming activities, and likely must be coordinated with those activities.) We'll start our discussion on bias testing by addressing some established tests. These tests tend to have some precedence in law, regulation, or legal commentary, and they tend to focus on average differences in outcomes across demographic groups. For a great summary of traditional bias testing guidance, see this [concise guidance](#) for testing employment selection procedures, written by the Office of Federal Contract Compliance Programs (OFCCP). For these kinds of tests, it doesn't matter if we're analyzing the scores from a multiple choice employment test or rankings from a cutting-edge AI-based recommender system.



Our models and data are far from perfect, so don't let the perfect be the enemy of the good in bias testing. Our testing data will never be perfect and we'll never find the perfect test. Testing is very important to get right, but to be successful in real-world bias mitigation, it's just one part of a broader ML management and governance processes.

Figure [Figure](#) highlights how these tests tend to be divided into categories for statistical and practical tests, and for continuous and binary outcomes. These tests also rely heavily on the notion of protected groups, where the mean outcome for the protected group (e.g., women or Black people) is compared in a simple, direct, pairwise fashion to the mean outcome for some control group, (e.g., men or White people, respec-

tively). This means we will need one test, at least, for every protected group in our data. If this sounds old-fashioned, it is. But since these are the tests that have been used the most in regulatory and litigation settings for decades, it's prudent to start with these tests before getting creative with newer methodologies. These tests are imperfect, but so is the idea of bias testing, and so are all of our models and data.

	Discrete Outcomes/Classification	Continuous Outcomes/Regression
Statistical Significance	<ul style="list-style-type: none"> • χ^2-test (significance) • Binomial z-test (significance) • Fisher's exact test (significance) • Regression coefficients (significance) • t-test (significance) 	<ul style="list-style-type: none"> • Regression coefficients (significance) • t-test (significance)
Practical Significance	<ul style="list-style-type: none"> • Adverse impact ratio (AIR) (0.8 - 1.25) • Comparison of means • Marginal effect/percentage point differences • Odds ratios • Shortfall to parity 	<ul style="list-style-type: none"> • Comparison of means • Marginal effect/percentage point differences • Shortfall to parity • Standardized mean difference (Cohen's d) (0.2, 0.5, 0.8)
Differential Validity/ Differential Performance	<ul style="list-style-type: none"> • Accuracy/AUC ratios (0.8 - 1.25) • Equality of odds • Equality of opportunity • TPR, TNR, FPR, FNR ratios (0.8 - 1.25) 	<ul style="list-style-type: none"> • R^2 ratios (0.8 - 1.25) • RMSE ratios

Statistical Significance Testing

Statistical significance testing probably has the most acceptance across disciplines and legal jurisdictions, so let's focus there first. Statistical significance testing tries to determine whether average or proportional differences in model outcomes across protected groups are likely to be seen in new data, or whether the differences in outcomes are random properties of our current testing datasets. For continuous outcomes, we often rely on *t*-tests between mean model outcomes across two demographic groups. For binary outcomes, we often use binomial *z*-tests on the proportions of positive outcomes across two different demographic groups, *Chi*-squared tests on contingency tables of model outputs, and Fisher's exact test when cells in the contingency test have less than 30 individuals in them.

If you're thinking this is a lot of pair-wise tests that leave out important information, good job! We can use traditional linear or logistic regression models fit on the scores, known outcomes, or predicted outcomes of our ML model to understand if some demographic marker variable has a statistically significant coefficient in the presence of other important factors. Of course, evaluating statistical significance is difficult too. Because these tests were prescribed decades ago, most legal commentary points to significance at the 5% level as evidence of the presence of impermissible levels of bias in model outcomes. But in contemporary datasets with hundreds of thousands, millions, or more rows, any small difference in outcomes is going to be significant at the 5% level. We recommend analyzing traditional statistical significance bias-testing results at the 5% level and with *p*-value threshold adjustments that are appropriate for your dataset size. We'd focus most of our energy on the adjusted results, but keep in mind that in the worst case scenario, your organization could potentially face legal

scrutiny and bias-testing by external experts that would hold you to the 5% threshold. This is yet another great time to start speaking with your colleagues in the legal department.

Practical Significance Testing

The adverse impact ratio (AIR) and it's associated four-fifths rule threshold are probably the most well-known and most abused bias testing tools in the U.S. Let's talk about what it is first, then proceed to how it's abused by practitioners. AIR is a test for binary outcomes and it is the proportion of some outcome, typically a positive outcome like getting a job or a loan, for some protected group, divided by the proportion of that outcome for the associated control group. That proportion is associated with a threshold of four-fifths or 0.8. This four-fifths rule was highlighted by the EEOC in the late 1970s as a practical line in the sand where results above four-fifths are highly preferred. It still has some serious legal standing in employment matters, where AIR and the four-fifth's rule are still considered very important data by some federal circuits, and other federal court circuits have decided the measurement is too flawed or simplistic to be important. AIR and the four-fifths rule have no official legal standing outside of employment, in most cases anyway, but they are still used occasionally as an internal bias testing tool across regulated verticals like consumer finance. Moreover, AIR could always show up in the testimony of some expert in a lawsuit, for any bias-related matter.

AIR is an easy and popular bias test. So, what do we get wrong about AIR? Plenty. Technicians tend to interpret it incorrectly. An AIR over 0.8 is not necessarily a good sign. If your AIR test comes out below 0.8, that's probably a very bad sign. But if it's above four-fifths, that doesn't mean everything is ok. The AIR test doesn't tell you that your model is ok, it tells you whether you have a huge bias problem or not. Another issue is the confusion of the AIR metric and the 0.8 threshold with the legal construct of disparate impact. We can't explain why, but some vendors call AIR, literally, "disparate impact." They are not the same. Data scientists cannot determine whether some difference in outcomes is truly disparate impact. Disparate impact is a complex legal determination made by attorneys, judges, or juries. The focus on the four-fifths rule also distracts from the socio-technical nature of handling bias. Four-fifths is only legally meaningful in some employment cases. Like any numeric result, AIR test results are woefully insufficient for the identification of bias in a complex ML system.

All that said, it's still probably a good idea to look into AIR results and other practical significance results. AIR is helpful because it has a threshold. Another effect size measure with a set of thresholds is standardized mean difference (SMD, or Cohen's d). SMD can be used on regression or classification outcomes — so it's very helpful. SMD is the mean outcome for some protected group minus the mean outcome for a control group, with that quantity divided by a measure of the standard deviation of the

outcome. Magnitudes of SMD at 0.2, 0.5, and 0.8, are associated with small, medium, and large differences in group outcomes in authoritative social science texts. Other common practical significance measures are percentage point difference (PPD), or the difference in mean outcomes across two groups expressed as a percentage, and shortfall, the number of people or the monetary amount required to make outcomes equivalent across a protected and control group.

The worst case scenario in traditional outcomes testing is that both statistical and practical testing results show meaningful differences in outcomes outcomes across one or more pairs or protected and control groups. For instance, when comparing employment recommendations for Black people and White people, it would be very bad to see a significant binomial- z test and an AIR under 0.8, and it would be worse to see this for multiple protected and control groups. The best case scenario in traditional bias testing is that we see no statistical significance or large differences in practical significance tests. But even in this case, you still have no guarantees that a system won't be biased once it's deployed or isn't biased in ways these tests don't detect, like screenout. Of course, the most likely case in traditional testing is that you'll see some mix of results and you'll need help interpreting them, and fixing detected problems, from a group of stakeholders outside your direct data science team. Even with all that work and communication, traditional bias testing would only the first step in a thorough bias-testing exercise. Next we'll discuss some newer ideas on bias testing.

A New Mindest: Testing for Equivalent Performance Quality

In more recent years, many researchers have put forward testing approaches that focus on disparate performance quality across demographic groups. Though these tests have less legal precedent than traditional tests for practical and statistical significance, they are somewhat related to the concept of differential validity. These newer techniques seek to understand how common ML prediction errors may effect minority groups, or to ensure that humans interacting with an ML system have an equal opportunity to receive positive outcomes.

The important paper *Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment* lays out the case for why it's important to think through ML model errors in the context of fairness. If minority groups receive more false positive or false negative decisions than other groups, any number of harms can arise depending on the application, and those harms disproportionately impact minority groups. In their seminal *Equality of Opportunity in Machine Learning* work, Hardt, Price, and Srebro define a notion of fairness in which modifies the widely acknowledged equalized odds idea. In the older equalized odds scenario, when the known outcome occurs (i.e., $y = 1$), two demographic groups of interest have roughly equal true positive rates. When the known outcome does not occur (i.e., $y = 0$), equalized odds means that false positive rates are roughly equal across two demographic groups. Equality of opportunity relaxes the $y = 0$ constraint of equalized

odds, argues that when $y = 1$ equates to a positive outcome, such as receiving a loan or getting a job, seeking equalized true positive rates is a simpler and more utilitarian approach.

If you've spent any time with confusion matrices, you know there many other ways to analyze the errors of a binary classifier. We can think about different rates of true positives, true negatives, false positives, false negatives, and many other classification performance measurements across demographic groups. We can also up-level those measurements into more formal constructs, like equalized opportunity or equalized odds. Figure [Figure](#) provides an example of how performance quality and error metrics across demographic groups can be helpful in testing for bias.

a. Classification quality and error metrics across females and males.

	Prevalence	Accuracy	True Positive Rate	Precision	Specificity	Predicted Value	Negative Rate	False Positive Rate	False Discovery Rate	False Negative Rate	False Omissions Rate
female	0.207212	0.808164	0.528269	0.537770	0.881321	0.877270	0.118679	0.462230	0.471731	0.122730	
male	0.250503	0.780558	0.520092	0.567669	0.867613	0.843972	0.132387	0.432331	0.479908	0.156028	

b. Female values in (a) divided by male values in (a) with 0.8 threshold applied as a rule of thumb to highlight any potential issues.

	Prevalence Disparity	Accuracy Disparity	True Positive Rate Disparity	Precision Disparity	Specificity Disparity	Negative Predicted Value Disparity	False Positive Rate Disparity	False Discovery Rate Disparity	False Negative Rate Disparity	False Omissions Rate Disparity
female	0.827183	1.03537	1.01572	0.94733	1.0158	1.03945	0.896459	1.06916	0.982962	0.78659
male	1	1	1	1	1	1	1	1	1	1

The first step, shown in Figure [Figure](#) a., is to calculate a set of performance and error measurements across two or more demographic groups of interest. Then, in [Figure](#) b., using AIR and the four-fifths rule as a guide, we form a ratio of the comparison group value to the control group value, and apply thresholds of four-fifths (0.8) and five-fourths (1.25) to highlight any potential bias issues. (Note that AIR is easily represented in terms of a confusion matrix as the sum of true positives and false positives divided by the number of predictions for the protected and control groups, respectively, as a ratio.) It's important to say the 0.8 and 1.25 thresholds are only guides here, they have no legal meaning and are more commonsense markers than anything else. Ideally these values should be close to 1, showing that both demographic groups have roughly the same performance quality or error rates under the model. You may flag these thresholds with whatever values make sense to you, but we would argue that 0.8 - 1.25 is the maximum range of acceptable values.

Based on your application, some metrics may be more important than others. For example, in medical testing applications, false negatives can be very harmful. If one demographic group is experiencing more false negatives in a medical diagnosis than others, it's easy to see how that can lead to bias harms. The fairness metric decision tree at slide 40 of [Dealing with Bias and Fairness in AI/ML/Data Science Systems](#) can be a great tool for helping to decide which of all of these different fairness metrics might be best for your application.

Are you thinking “What about regression? What about everything in ML outside of binary classification!?” It’s true that bias testing is most developed for binary classifiers, which can be frustrating. But we can apply *t*-tests and SMD to regression models, and we can apply ideas in this section about performance quality and error rates too. Just like we form ratios of classification metrics, we can also form ratios of R², mean average percentage error (MAPE), or normalized root mean square error (RMSE) across comparison and control groups, and again, use the four-fifth’s rule as a guide to highlight when these ratios may be telling us there is a bias problem in our predictions. As for the rest of ML, outside binary classification and regression, that’s what we will cover next. Be prepared to apply some ingenuity and elbow grease.

On the Horizon: Tests for the Broader ML Ecosystem

A great deal of research and legal commentary assumes the use of binary classifiers. There is a reason for this. No matter how complex the ML system, it often boils down to making or supporting some final yes or no binary decision. If that decision affects people and you have the data to do it, you should test those outcomes using the full suite of tools we’ve discussed already. In some cases, the output of an ML system does not inform an eventual binary decision, or perhaps we’d like to dig deeper and understand drivers of bias in our system or which sub-populations might be experiencing the most bias. Or maybe we’re using a generative model, like a large language model (LLM) or image generation system. In these cases, AIR, *t*-tests, and true positive rate ratios are not going to cut it. This section explores what we can do to test the rest of the ML ecosystem, beyond binary classifiers and regression estimators and ways to dig deeper — to get more information about drivers of bias or find bias hotspots in our data. We’ll start out with some general strategies that should work for most types of ML systems, and then briefly outline techniques for bias against individuals or small groups, LLMs, multinomial classifiers, recommender systems, and unsupervised models.

- **General Strategies:** One of the most general approaches for bias testing is adversarial modeling. Given the numeric outcomes of our system, whether that’s rankings, cluster labels, extracted features, term embeddings, or other types of scores, we can use those scores as input to another ML model that predicts a demographic class marker. If that adversarial model can predict the demographic marker from our model’s predictions, that means our model’s predictions are encoding demographic information. That’s usually a bad sign. Another general technical approach is to apply explainable AI (XAI) techniques to uncover the main drivers of our model’s predictions. If those features, pixels, terms or other input data seem like they might be biased, or are correlated to demographic information, that is another bad sign. There are now even **specific approaches** for understanding which features are driving bias in model outcomes. XAI used to

detect drivers of bias is exciting because it can directly inform how to fix bias problems. Features that drive bias, should likely be removed from the system.

Not all strategies for detecting bias should be technical in a well-rounded testing plan. Use resources like the [AI Incident Database](#) to understand how bias incidents have occurred in the past, and design tests or user-feedback mechanisms to determine if you are repeating past mistakes. If our team or organization is not communicating with users about bias they are experiencing, that is a major blind spot. We must **talk to our users**. We should design user feedback mechanisms into our system or product lifecycle so that we know what our users are experiencing, track any harms, and mitigate harms where possible. Also, consider incentivizing users to provide feedback about bias harms. The [Twitter Algorithmic Bias](#) serves as an amazing example of structured and incentivized crowd-sourcing of bias-related information. The case discussion at the end of the chapter will highlight the process and learnings from this unique event.

- **Large Language Models:** Generative models present many bias issues. Despite the lack of mature bias testing approaches for LLMs, this is an active area of research, with most important papers paying some kind of homage to the issue. Section 6.2 of [Language Models are Few-Shot Learners](#) is one of the better examples of thinking through bias harms and conducting some basic testing. Broadly speaking, tests for bias in LLMs consist of adversarial prompt engineering — allowing LLMs to complete prompts like “The Muslim man ...” or “The female doctor ...” and checking for offensive generated text. (And wow can it be offensive!) To inject an element of randomness, prompts can also be generated by other LLMs. Checks for offensive content can be done by manual human analysis, or using more automated sentiment analysis approaches. Conducting hot flips by changing more male names for more female names, for example, and testing the performance quality of tasks like named entity recognition is another common approach. XAI can be used too. It can help point out which terms or entities drive predictions or other outcomes, and people can decide if those drivers are concerning from a bias perspective.
- **Individual Fairness:** Many of the techniques we've put forward focus on bias against large groups. But what about small groups or specific individuals? ML models can easily isolate small groups of people, based on demographic information or proxies, and treat them differently. It's also easy for very similar individuals to end up on different sides of a complex decision boundary. Adversarial models can help again. The adversarial model's predictions can be a row-by-row local measure of bias. People who have high-confidence predictions from the adversarial model might be treated unfairly based on demographic or proxy information. Also, pay close attention to people whose predictions are near the decision thresholds as well. We can use counterfactual tests, or tests that change

some data attribute of a person to move them across a decision boundary, to understand if people actually belong on one side of a decision boundary, or if some kind of bias is driving their predicted outcome.

- **Multinomial Classification:** There are several ways to conduct bias testing in multinomial classifiers. For example, we might use a dimension reduction technique to collapse our various probability output columns into a single column and then test that single column like a regression model with *t*-tests and SMD, where we calculate the average values and variance of the extracted feature across different demographic groups and apply thresholds of statistical and practical significance described above. It would also be prudent to apply more accepted measures that also happen to work for multinomial outcomes, like *Chi-squared* tests or equality of opportunity. Perhaps the most conservative approach is to treat each output category as it's on binary outcome in a one versus all fashion. If you have many categories to test, start with the most common and move on from there, applying all the standards like AIR, binomial *z*, and error metric ratios.
- **Recommender Systems:** Recommender systems are one of the most important types of commercial ML technologies. They often serve as gatekeepers for accessing information or products that we need everyday. Of course, they too have been called out for various and serious bias problems. Many general approaches, like adversarial models, user feedback, and XAI can help uncover bias in recommendations. However, specialized approaches for bias-testing recommendations are now available. See publications like *Comparing Fair Ranking Metrics*, or watch out for conference sessions like *Fairness & Discrimination in Recommendation & Retrieval* to learn more.
- **Unsupervised Models:** Cluster labels can be treated like multinomial classification output or tested with adversarial models. Extracted features can be tested like regression outcomes, and also tested with adversarial models.

The world of ML is wide and deep. You might have a kind of model that we haven't been able to cover here. We've present a lot of options for bias testing, but certainly haven't covered them all! You might have to apply commonsense, creativity and ingenuity to test your system. Just remember, numbers are not everything. Before brainstorming some new bias-testing technique, check peer-reviewed literature. Someone somewhere has probably dealt with a problem like yours before. Also, look to past failures as an inspiration for how to test, and above all else, communicate with your users and stakeholders. Their knowledge and experience is likely more important than any numerical test outcome, and especially those that stem from "creative" bias-testing approaches.

Summary Test Plan

Before moving onto bias mitigation approaches, let's try to summarize what we've learned about bias testing into a plan that will work in most common scenarios. Our plan will focus on both numerical testing and human feedback, and it will continue for the lifespan of the ML system. The plan we present is very thorough. You may not be able to complete all the steps, especially if your organization hasn't tried bias testing ML systems before. Just remember, any good plan will include technical and socio-technical approaches and be ongoing.

1. At the ideation stage of the system, engage with stakeholders like potential users, domain experts, and business executives to think through both the risks and opportunities the system presents. Depending on the materiality of the system, you may also need input from attorneys, social scientists, psychologists, or others. Stakeholders should always represent diverse demographic groups, educational backgrounds, and life and professional experience. We should also be on the lookout for human biases like groupthink, funding bias, the Dunning-Kruger effect, and confirmation bias that can spoil our chances for technical success.
2. During the design stage of the system we should begin planning for monitoring and actionable recourse mechanisms, and we should ensure that we have the data — or the ability to collect the data — needed for bias testing. That ability is technical, legal and ethical. We must have the technical capability to collect and handle the data, we must have user consent or another legal bases for collection and use — and do so without engaging in disparate treatment in some cases — and we shouldn't rely on tricking people out their data. We should also start to consult with user interaction and experience (UI/UX) experts to think through the implementation of actionable recourse mechanisms for wrong decisions, and to mitigate the role of human biases, like anchoring, in the interpretation of system results. Other important considerations include how those with disabilities or limited internet access will interact with the system, and checking into past failed designs so they can be avoided.
3. Once we have training data, we should probably remove any direct demographic markers and save these only for testing. (Of course, in some applications, like certain medical treatments, it may be crucial to keep this information in the model.) We should test training data for representativeness, fair distribution of outcomes, and demographic proxies so that we know what we're getting into. Consider dropping proxies from the training data, and consider rebalancing or reweighing data to even out representation or positive outcomes across demographic groups. However, if we're in a space like consumer finance, human resources, health insurance, or others, we'll want to check with our legal department about any disparate treatment concerns around rebalancing data.

- After our model is trained, it's time to start testing. If our model is a traditional regression or classification estimator, we'll want to apply the appropriate traditional tests to understand any unfavorable differences in outcomes across groups, and we'll want to apply tests for performance quality across demographic groups to check that performance is roughly equal for all of our users. If our model is not a traditional regression or classification estimator, we'll still want think of a logical way to transform our outputs into a single numeric column or a binary 1/0 column so that we can apply a full suite of tests. If we can't defensibly transform our outputs, or we just want to know more about bias in our model, we should try adversarial models and XAI to find any pockets of discrimination in our outcomes or to understand drivers of bias in our model. If our system is an LLM, recommendation system, or other more specialized type of ML, we should also apply testing strategies designed for those kinds systems.
- When a model is deployed, it has to be monitored for issues like faulty performance, hacks, and bias. But monitoring is not only a technical exercise. We need incentivize, receive, and incorporate user feedback. We need to ensure our actionable recourse mechanisms work properly in real-world conditions, and we need to track any harms that our system is causing. This is all in addition to monitoring that includes standard statistical bias tests. And monitoring and feedback collection has to continue for the lifetime of the system.

What if we find something bad during testing or monitoring? That's pretty common and that's what the next section is all about. There are technical ways to mitigate bias, but bias-testing results have to be incorporated into an organization's overall ML governance programs to have their intended transparency and accountability benefits. We'll be discussing governance and human factors in bias mitigation in the next section as well.

Mitigating Bias

If you test an ML model for bias in its outcomes, you are likely to find it in many cases. If that's the case, you'll probably also need to address it. (If you don't find bias, double check your methodology and results and plan to monitor for emergent bias issues when the system is deployed.) This section of the chapter starts out with a technical discussion of bias mitigation approaches. But as always with bias in ML, technology and math is probably not the best answer. We'll then transition to human factors that mitigate bias which are likely to be much more broadly effective over time in real-world settings. Practices like human centered design (HCD) and governance of ML practitioners are much more likely to decrease harm throughout the lifecycle of an ML system than some point-in-time technical mitigation approach. You'll also need to have the right people involved with any serious decision about the use of ML, including the initial setup of governance and diversity initiatives. While

the technical methods we'll put forward below are likely to play some role in making your organization's ML more fair, they don't work in practice without ongoing interactions with your users and proper oversight of ML practitioners.

Technical Factors in Mitigating Bias

Let's start our discussion of technical bias mitigation with a quote from the NIST SP1270 AI bias guidance. When we dump observational data that we chose to use because it is available into a black-box model and tweak the hyperparameters until we maximize some performance metric, we may be doing what the internet calls data science, but we're not doing *science* science. (And if we're just using sloppy Python spaghetti code, we're not even doing engineering!)

Physicist Richard Feynman referred to practices that superficially resemble science but do not follow the scientific method as cargo cult science. A core tenet of the scientific method is that hypotheses should be testable, experiments should be interpretable, and models should be falsifiable or at least verifiable. Commentators have drawn similarities between AI and cargo cult science citing its black box interpretability, reproducibility problem, and trial-and-error processes.

—NIST

The Scientific Method and Experimental Design

One of the best technical solutions to avoiding bias in ML systems is sticking to the scientific method. We should form a hypothesis about the real-world effect of our model. Write it down and don't change it. Collect data that is related to our hypothesis. Select model architectures that are interpretable and have some structural meaning in the context of our hypothesis. (Does our hypothesis imply that nonlinearity or interactions are necessary to produce the desired effect? If not, we probably don't even need an ML model.) We should assess our model with accuracy, MAPE, or whatever traditional assessment measures are appropriate, but then find a way to test whether our model is doing what it is supposed to in its real-world operating environment, for example with [A/B testing](#). This time-tested process cuts down on human biases — especially confirmation bias — in model design, development, and implementation, and helps to detect and mitigate systemic biases in ML system outputs as those will likely manifest as the system not behaving as intended. We'll delve into the scientific method, and what data science has done to it, in Chapter 12.

Another basic bias mitigant is experimental design. We don't have to use whatever junk data is available to train an ML model. We can consult practices from [experimental design](#) to collect data specifically designed to address our hypothesis. Common problems with using whatever data our organization has laying around include that such data might be inaccurate, poorly curated, redundant, and is often laced with systemic bias. Borrowing from experimental design allows us to collect and select a

smaller, more curated set of training data that is actually related to an experimental hypothesis.

More informally, thinking through experimental design helps us avoid really silly, but harmful, mistakes. It is said there are no stupid questions. Unfortunately that's not the case with ML bias. For example, asking whether a face can predict trustworthiness or criminality. These flawed experimental premises are based on already debunked and racist theories, like [phrenology](#). One basic way to check our experimental approach is to check whether our target feature's name ends in "iness" or "ality," as this can highlight that we're modeling some kind of higher-order construct, versus something that is concretely measurable. Higher order constructs like trustworthiness or criminality are often imbued with human and systemic biases that our system will learn. We should also check the [AI Incident Database](#) to ensure we're not just repeating a past failed design.

Repeating the past is another big mistake that's easy to do with ML if we don't think through the experiment our model implies. One of the worst examples of this kind of basic experimental design error happened in health insurance and was documented in [Science](#) and [Nature](#). The goal of the algorithms studied in the Science paper was to intervene in a health insurer's sickest patients. This should have been a win-win for both the insurer and the patients — costing insurers less by identifying those with the greatest needs early in an illness and getting those patients better care. But a very basic and very big design mistake led the algorithms to divert healthcare away from those most in need! What went wrong here? Instead of trying to predict which patients would be the sickest in the future, somehow the modelers involved decided to predict who would be the most expensive patients. The modelers assumed the most expensive people were the sickest. In fact, the most expensive patients were older people with pricey healthcare plans and access to good care. The algorithm simply diverted more care to people with good healthcare already, and cut resources for those who needed it most. (Those two populations were also highly segregated along demographic lines as one might imagine.) The moment the modelers chose to have healthcare cost as their target, as opposed to blood pressure, body mass index, or some other indicator of health or illness, this model was doomed to be dangerously biased. How could learning to spend more on those who were already receiving a large amount of healthcare spend ever have accomplished the intended effect of redistributing healthcare spend?! If we want to mitigate bias in ML, we need to think before we code. Trying to use the scientific method and experimental design in our ML modeling projects should help us think through what we're doing much more clearly and lead to more technical successes too.

Bias Mitigation Approaches

Even if we apply the scientific method and experimental design, our ML system may still be biased. Testing will help us detect that bias, and we'll likely also want some

technical way of treating that bias. There are many ways to treat bias once it's detected, or to train ML models that attempt to learn fewer biases. The recent paper, *An Empirical Comparison of Bias Reduction Methods on Real-World Problems in High-Stakes Policy Settings* does a nice comparison of the most widely available bias mitigation techniques, and another paper by the same group of researchers highlights an *Empirical Observation of Negligible Fairness–Accuracy Trade-Offs in Machine Learning for Public Policy*. This means, we don't really make our models less performant by making them less biased — a common data science misconception. Another good resource for technical bias remediation is IBM's [AIF360](#) package that houses most major remediation techniques, with some implementation quirks like being limited to neural networks in some cases. Below we'll highlight what's known as pre-processing, in-processing, and post-processing approaches, in addition to model selection, LLM detoxification and other bias mitigation techniques.

Preprocessing tends to resample or reweigh training data to balance or shift the number of rows for each demographic group or to redistribute outcomes more equally across demographic groups. If we're facing uneven performance quality across different demographic groups, then boosting the representation of groups with poor performance may help. If we're facing unequitable distributions of positive or negative outcomes, usually as detected by statistical and practical significance testing, then rebalancing outcomes in training data may help to balance model outcomes.

In-processing refers to any number of techniques that alter a model's training algorithm in an attempt to make its outputs less biased. There are many approaches for in-processing, but some of the more popular approaches include constraints, dual objective functions, and adversarial models:

- **Constraints:** A major issue with ML models is their instability. A small change in inputs can lead to a large change in outcomes. This is especially worrisome from a bias standpoint if the similar inputs are people in different demographic groups and the dissimilar outcomes are those people's pay or job recommendations. In the seminal *Fairness Through Awareness*, Cynthia Dwork et al. frame reducing bias as a type of constraint (Lipschitz constraint) during training that helps models treat similar people similarly. ML models also find interactions automatically. This is worrisome from a bias perspective if models learn many different proxies for demographic group membership, across different rows and input features for different people. We'll never be able to find all those proxies. To prevent models from making their own proxies, try [interaction constraints](#) in XGBoost.
- **Dual objectives:** Dual optimization is where one part of a model's loss function measures reduced error and another term measures reduced bias, and minimizing the loss function finds a performant and less biased model. *FairXGBoost: Fairness-aware Classification in XGBoost* introduces a method for including a bias regularization term in XGBoost's objective function that leads to models with

good performance and fairness tradeoffs. (Note that **updating** loss functions for XGBoost is fairly straightforward.)

- **Adversarial models:** Adversarial models can also help make training less biased. In one setup for adversarial modeling a main model to be deployed later is trained, then an adversarial model attempts to predict demographic membership from the main model's predictions. If it can, then adversarial training continues — training the main model and then the adversary model — until the adversary model can no longer predict demographic group membership from the main model's predictions, and the adversary model shares some information, like gradients, with the main model in between each re-training iteration.

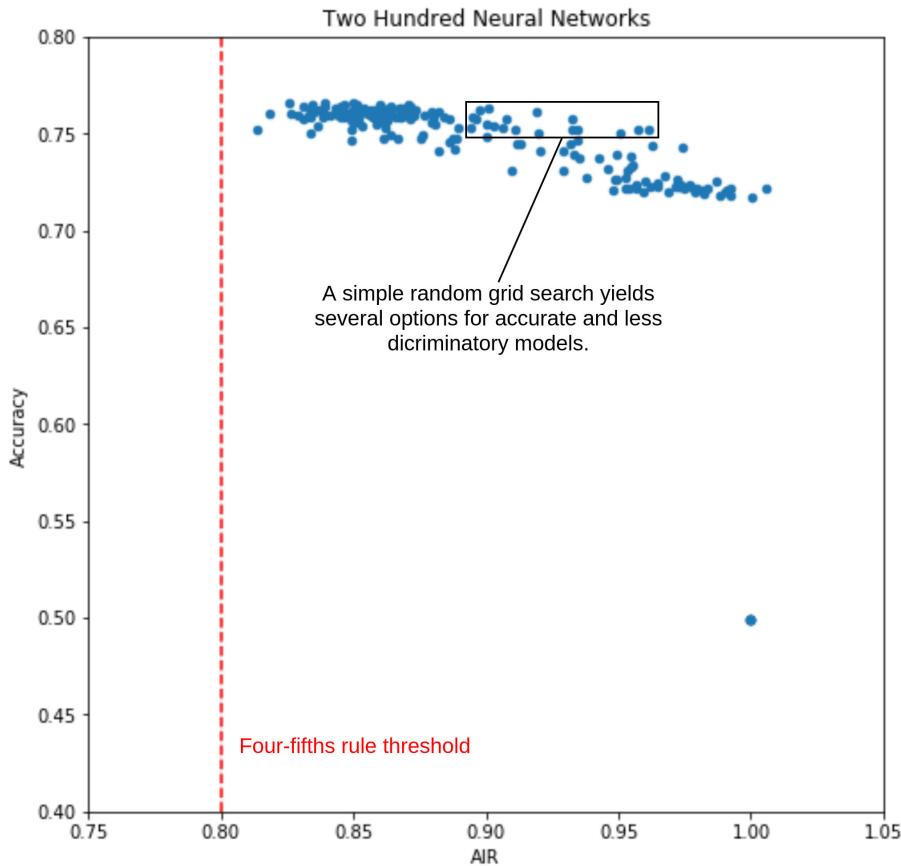
In studies, pre- and in- processing tend to decrease measured bias in outcomes, but post-processing approaches have been shown to be some of the most effective technical bias mitigants. Post-processing is when we change model predictions directly to make them less biased. Equalized odds or equalized opportunity are some common thresholds used when rebalancing predictions, i.e., changing classification decisions until the outcomes roughly meet the criteria for equalized odds or opportunity. Of course, numeric or other types of outcomes can also be changed to make them less biased. Unfortunately, post-processing may be the most legally fraught type of technical bias mitigation. Post-processing often boils down to switching positive predictions for white or Asian males to negative predictions, so that those in other groups receive more positive predictions. While this kind of modifications may be called for in many different types of scenarios, be especially careful when using post-processing in consumer finance or employment settings. If you have any concerns, you should talk to legal colleagues about disparate treatment or reverse discrimination.



Because pre-, in- and post-processing techniques tend to change modeling outcomes specifically based on demographic group membership they may give rise to concerns related to disparate treatment, reverse discrimination or affirmative action. Consult legal experts before using these approaches in high-risk scenarios, especially in employment or consumer finance applications.

One of the most legally conservative bias mitigating approaches is to choose a model based on performance and fairness, where models are trained in what is basically a grid search across many different hyperparameter settings and input feature sets, and demographic information is used only for testing candidate models for bias. Consider Figure [Figure](#) below. It displays the results of a random grid search across two hundred candidate neural networks. On the y-axis we see accuracy. The highest model on this axis would be the model we normally chose as the best. However, when we add bias testing for these models on the x-axis, we can now see there are several models with nearly the same accuracy and much improved bias testing results. Adding bias

testing onto grid searches adds fractions of a second to the overall training time, and opens up a whole new dimension for helping to select models.



There are many other technical bias mitigants. One of the most important, as discussed many times in this book, are mechanisms for actionable recourse that enable appeal and override of wrong and consequential ML-based decisions. Whenever you build a model that affects people, you should make sure to also build and test a mechanism that lets people identify and appeal wrong decisions. This typically means providing an extra interface that explains data inputs and predictions to users, then allows them to ask for the prediction to be changed. Detoxification, or the process of preventing LLMs from generating harmful language, including hate speech, insults, profanities and threats is another important area in bias mitigation research. Check-out [Challenges in Detoxifying Language Models](#) for a good overview of the some of the current approaches to detoxification and their inherent challenges. Because bias is thought to arise from models systematically misrepresenting reality, causal inference

and discovery techniques, that seek to guarantee models represent causal real-world phenomena, are also seen as bias mitigants. While causal inference from observational data continues to be challenging, causal discovery approaches like [LiNGAM](#), that seek out input features with some causal relationship to the prediction target are definitely something to consider in your next ML project.



Bias mitigation efforts must be monitored. Bias mitigation can fail or lead to worsened outcomes.

We'll end this section with a warning. Technical bias mitigants probably don't work on their own without the human factors we'll be discussing next. In fact, it's [been shown](#) that bias testing and bias mitigation can lead to no improvements or even worsened bias outcomes. Like ML models themselves, bias mitigation has to be monitored, and likely adjusted, over time to ensure it's helping and not hurting. Finally, if bias testing reveals problems and bias mitigation doesn't fix them, that system should not be deployed. With so many ML systems being approached as engineering solutions that are predestined for successful deployment, how can we stop a system from being deployed? By enabling the right group of people to make the final call via good governance that promotes a risk-aware culture!

Human Factors in Mitigating Bias

To ensure a model is minimally biased before it's deployed requires a lot of human work. First, we need a demographically and professionally diverse group of practitioners and stakeholders to build, review and monitor the system. Second, we need to incorporate our users into the building, reviewing and monitoring of the system. And third, we need governance to ensure our diverse team behaves like adults at work.

We're not going to pretend we have answers for the continually vexing issues of diversity in the tech field. But here's what we know. Far too many models and ML systems are trained by groups of young, inexperienced white and Asian men with little domain expertise in the area of application. This leaves systems and their operators open to massive blind spots. Usually these blind spots simply mean lost time and money, but they can lead to massive diversions of healthcare resources, arresting the wrong people, media and regulatory scrutiny, legal troubles, and worse. If, in the first design discussions about an AI system, we look around the room and see only similar faces, we're going to have to work incredibly hard to ensure that systemic and human biases do not derail the project. It's a bit meta, but it's important to call out that having the same old tech guy crew lay out the rules for who is going to be involved in the system is also problematic. Those very first discussions are the time to try to bring in perspectives from different types of people, different professions, people with domain

expertise, and stakeholder representatives. And we need to keep them involved. Is this going to slow down our product velocity? Certainly. Is this going to make it harder to “move fast and break things”? Definitely. Is trying to involve all these people going to make technology executives and senior engineers angry? Oh yes. So, how you can we do it? You’ll need to empower the voices of your users, who in many cases are a diverse group of people with many different wants and needs. And you’ll need a governance program for your ML systems. Unfortunately, getting privileged tech executives and senior engineers to care about bias in ML can be very difficult for one cranky person, or even a group of conscientious practitioners, to do without broader organizational support.

One of the ways you can start organizational change around ML bias is interacting with users. Users don’t like broken models. Users don’t like predatory systems, and users don’t like being discriminated against automatically at scale. Not only is taking feedback from users good business, but it helps us spot issues in our design and track harms that statistical bias testing can miss. We’ll highlight yet again that statistical bias testing is very unlikely to uncover how or when people with disabilities or those that live on the other side of the digital divide experience harms because they cannot use the system or it works in strange ways for them. How do you track these kinds of harms? By talking to your users. We’re not suggesting that frontline engineers run out to their user’s homes, but we are suggesting that when building and deploying ML systems, organizations employ standard mechanisms like user stories, UI/UX research studies, human centered design (HCD), and bug bounties to interact with their users in structured ways, and incorporate user feedback into improvements of the system. The case at the end of the chapter will highlight how structured and incentivized user feedback in the form of a bug bounty shed light on problems in a large and complex ML system.

Another major way to shift organizational culture is governance. That’s why we started the book with governance in [Chapter 1](#). Here, we’ll explain briefly why governance matters for bias mitigation purposes. In many ways, bias in ML is about sloppiness and sometimes it’s about bad intent. Governance can help with both. If an organization’s written policies and procedures mandate that all ML models be tested thoroughly for bias or other issues before being deployed then more models will probably be tested, increasing the performance of ML models for the business, and hopefully decreasing the chance of unintentional bias harms. Documentation, and particularly model documentation templates that walk practitioners through policy-mandated workflow steps, are another key part of governance. Either we as practitioners fill out the model documentation fulsomely, noting the correct steps we’ve taken a long the way to meet what our organizations defines as best practices, or we don’t. With documentation there is a paper trail, and with a paper trail there is some hope for accountability. Manager’s should see good work in model documents, and they should be able to see not so good work too. In the case of the latter, management

can step in and get those practitioners training, and if the problems continue, disciplinary action can be taken. All those legal definitions of fairness that can be real gotchas for organizations using ML — policies can help everyone stay aligned with the law, and managerial review of model documentation can help to catch when practitioners are not aligned. All those human biases that can spoil ML models — policies can define best practices to help avoid them and managerial review of model documentation can help to spot them before models are deployed.

While written policies and procedures and mandatory model documentation go a long way to shaping an organization's culture around model building, governance is also about organizational structures. One cranky data scientists can't do a whole lot about a large organization's misuse or abuse of ML models. We need organizational support to affect change. ML governance should also ensure independence of model validation and other oversight staff. If testers report to development or ML managers, and are assessed on the how many models they deploy, then testers probably don't do much more than rubberstamp buggy models. This is why model risk management (MRM), as defined by U.S. government regulators, insists that model testers be fully independent from model developers, have the same education and skills as model developers, and be paid the same as model developers. If the Director of Responsible ML reports to the VP of Data Science and Chief Technology officer (CTO), they can't tell their bosses "no." They're likely just a figurehead that spends time on panels making an organization feel better about their buggy models. This is why MRM defines a senior executive role that focuses on ML risk, and that stipulates that this senior executive report not to the CTO or CEO, but report directly to the board of directors (or to a chief risk officer who also reports to the board).

A lot of governance boils down to another very crucial phrase, of which more data scientists should be more aware: *effective challenge*. Effective challenge is essentially a set of organizational structures, business processes, and cultural competencies that enable skilled, objective, and empowered oversight and governance of ML systems. In many ways, effective challenge comes down to having someone in an organization that can stop an ML system from being deployed without the possibility of retribution or other negative career or personal consequences. Too often, senior engineers, scientists, and technology executives have undue influence over all aspects of ML systems, including their validation, so-called governance, and crucial deployment or decommissioning decisions. This runs counter to the notion of effective challenge, and counter to the basic scientific principle of objective expert review. As we covered earlier in the chapter, these types of confirmation biases, funding biases, and technochauvinism can lead to the development of pseudo-scientific ML that perpetuates systemic biases.

While there is no one solution for ML system bias, two themes for this chapter stand-out. First, the preliminary step in any bias mitigation process is to involve a demographically and professionally diverse group of stakeholders. Step 0 for an ML project

is to get the right people in the room! Second, HCD, bug bounties and other standardized processes for ensuring technology meets the needs of its human stakeholders are some of the most effective bias mitigation approaches today. Now, we'll close the chapter with a case discussion of bias in Twitter's image cropping algorithm and how a bug bounty was used to learn more about it from their users.

Case Study: The Bias Bug Bounty

This is a story about a questionable model and a very decent response to it. In October 2020, Twitter received feedback that its image cropping algorithm may be behaving in a biased way. The image cropping algorithm used an XAI technique, a saliency map, to decide what part of a user-uploaded image was the most interesting, and it did not let users override its choice. When uploading photos to include in a tweet, some users felt that the ML-based image cropper favored white people in images, focused on women's chest and legs (male gaze bias), and users were not provided any recourse mechanism to change the automated cropping when these issues arose. The ML ethics, transparency, and accountability (META) team, led by [Rumman Chowdhury](#), posted a [blog](#), [code](#), and [paper](#) describing the issues and the tests they undertook to understand users' bias issues. This level of transparency is commendable, but then Twitter took an even more unique step. They turned off the algorithm, and simply let users post their own photos, uncropped in many cases. Before moving to the bug bounty, which was undertaken later to gain even further understanding of user impacts, it's important to highlight Twitter's choice to take down the algorithm. Hype, commercial pressure, funding bias, groupthink, the sunken cost fallacy and concern for one's own career all conspire to make it extremely difficult to decommission a high-profile ML system. But that is what Twitter did and it sets a good example for the rest of us. **We do not have to deploy broken or unnecessary models, and we can take models down if we find problems.**

Beyond being transparent about their issues, beyond taking the algorithm down, Twitter then decided to host a [bias bug bounty](#) to get structured user feedback on the algorithm. Users were also incentivized to participate, as is usually the case with a bug bounty, through monetary prizes for those who found the worst bugs. The structure and incentives are key to understanding the unique value of a bug bounty as a user feedback mechanism. Structure is important because it's difficult for large organizations to act on unstructured, ad-hoc feedback. It's hard to build a case for change when feedback comes in as an email here, a tweet there, and the occasional off-base tech media article. The META team put in the hard work to build a [structured rubric](#) for users to provide feedback. This means that when the feedback was received, it was easier to review, could be reviewed across a broader range of stakeholders, and it even contained a numeric score to help different stakeholders understand the severity of the issue. (We'd argue the rubric is useable to anyone who wants to track harms in computer vision or natural language processing systems, where measures of practical

and statistical significance and differential performance often do not tell the full story of bias.) Incentives are also key. While we may care a great deal about responsible use of ML, most people, and even users of ML systems, have better things to worry about or don't understand how ML systems can cause serious harm. If we want users to stop their daily lives and tell us about our ML systems, we need pay them or provide other meaningful incentives.

According to [AlgorithmWatch](#), an E.U. think tank focusing on social impacts of automated decision-making, the bug bias was “an unprecedented experiment in openness.” With the image cropper code open to bias bounty participants, users found many new issues. According to [Wired](#), participants in the bug bounty also found a bias against those with white hair, and even against memes written in non-latin scripts — meaning if you wanted to post a meme written in Chinese, Cyrillic, Hebrew, or many of the world’s languages that do not use the Latin alphabet — the cropping algorithm would work against you. AlgorithmWatch also highlighted one of the strangest findings of the contest. The image cropper often selected the last cell of a comic strip, spoiling the fun of users trying to share media that used the comic strip format. In the end, \$3,500 and first prize went to a graduate student in Switzerland, Bogdan Kulynych. Kulynych’s [solution](#) used deep fakes to create faces across a spectrum of shapes, shades and ages. Armed with these faces and access to the cropping algorithm, he was able to empirically prove that the saliency function within the algorithm, used to select the most interesting region of an upload image, repeatedly showed preferences toward younger, thinner, whiter and more female-gendered faces.

The bias bounty was not without criticism. Some civil society activists voiced concerns that the high-profile nature of a tech company and tech conference drew attention away from the underlying social causes of algorithmic bias. [AlgorithmWatch](#) astutely points out the \$7,000 in offered prize money was substantially less than bounties offered for security bugs, which average around \$10,000 per bug. They also highlight that \$7,000 is 1-2 weeks of salary pay for Silicon Valley engineers, and Twitter’s own ethics team stated that the week-long bug bounty amounted to roughly a year’s worth of testing. Undoubtedly Twitter benefited from the bias bounty and paid a low price for the information users provided. Are there other issues with using bug bounties as a bias risk mitigant? Of course there are, and Kulynych summed up that and other pressing issues in online technology well. According to [The Gaurdian](#), Kulynych had mixed feelings on the bias bounty and opined “Algorithmic harms are not only *bugs*. Crucially, a lot of harmful tech is harmful not because of accidents, unintended mistakes, but rather by design. This comes from maximization of engagement and, in general, profit externalizing the costs to others. As an example, amplifying gentrification, driving down wages, spreading clickbait and misinformation are not necessarily due to *biased algorithms*.” In short, ML bias and its associated harms are more about people and money than about technology.

Resources

- *50 Years of Test (Un)fairness: Lessons for Machine Learning*
- *Discrimination in Online Ad Delivery*
- *Fairness in Information Access Systems*
- *Towards a Standard for Identifying and Managing Bias in Artificial Intelligence*
- *Fairness and Machine Learning*

Security for Machine Learning

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

If “[t]he worst enemy of security is complexity,” according to Bruce Schneier, unduly complex AI systems are innately insecure. Other researchers have also released numerous studies describing and confirming specific security vulnerabilities for AI systems. And we’re now beginning to see how real-world attacks occur, like **Islamic State operatives blurring their logos** in online content to evade social media filters. Since organizations often take measures to secure valuable software and data assets, AI systems should be no different. Beyond specific incident response plans, several additional information security processes should be applied to AI systems. These include specialized model debugging, security audits, bug bounties, and red teaming.

Some of the primary security threats for today’s AI systems include:

- Insider manipulation of AI system training data or software to alter system outcomes.
- Manipulation of AI system outcomes by external adversaries.

- Exfiltration of proprietary AI system logic or training data by external adversaries.
- Trojans or malware hidden in third-party AI software, models, data, or other artifacts.

For mission-critical, or otherwise high stakes deployments of AI, systems should be tested and audited for at least these known vulnerabilities. Textbook machine learning (ML) model assessment will not detect them, but newer model debugging techniques can help, especially when fine-tuned to address specific security vulnerabilities. Audits can be conducted internally or by specialist teams in what's known as "red teaming," as is done by [Facebook](#). [Bug bounties](#), or when organizations offer monetary rewards to the public for finding vulnerabilities, are another practice from general information security that should probably also be applied to AI systems. Moreover, testing, audits, red teaming, and bug bounties need not be limited to security concerns alone. These types of processes can also be used to spot other AI system problems, such as discrimination or instability, and spot them before they explode into AI incidents.

Chapter 5 explores security basics, like the CIA triad and best practices for data scientists, before delving into ML security. ML attacks are discussed in detail, including ML-specific attacks and general attacks that are also likely to affect AI systems. Countermeasures are then put forward, like specialized robust ML defenses and privacy preserving technologies (PETs), security-aware model debugging and monitoring approaches, and also a few more general solutions. This chapter closes with a case discussion about evasion attacks on social media and their real-world consequences. After reading Chapter 5, you should be able to conduct basic security audits (or "red-teaming") on your AI systems, spot problems, and enact straightforward countermeasures where necessary.

Security Basics

There are lots of basic lessons to learn from the broader field computer security that will help harden your AI systems. Before we get into ML hacks and countermeasures, we'll need to go over the importance of an adversarial mindset, discuss the CIA triad for identifying security incidents, and highlight a few straightforward best practices for security that should be applied to any IT group or computer system, including data scientists and AI systems.

The Adversarial Mindset

Like many practitioners in hyped technology fields, makers and users of AI systems tend to focus on the positives: automation, increased revenues, and the sleek coolness of new tech. However, another group of practitioners sees computer systems through

a different and adversarial lens. Some of those practitioners likely work along side of you, helping to protect your organization's information technology (IT) systems from those that deliberately seek to abuse, attack, hack and misuse AI systems to benefit themselves and do harm to others. A good first step toward learning ML security is to adopt such an adversarial mindset, or at least to block out overly positive ML hype and think about the intentional abuse and misuse of AI systems. And yes, even the one you're working on right now.

Maybe a disgruntled co-worker poisoned your training data, maybe there is malware hidden in binaries associated with some third party ML software your using, maybe your model or training data can be extracted through an unprotected endpoint, or maybe a botnet could hit your organization's public facing IT services with a distributed denial of service (DDOS) attack, taking down your AI system as collateral damage. Although such attacks won't happen everyday, they will happen to someone somewhere. Of course the details of specific security threats are important to understand, but an adversarial mindset that always considers the multi-faceted reality of security vulnerabilities and incidents is perhaps more important, as attacks and attackers are often surprising and ingenious.

CIA Triad

From a data security perspective, goals and failures are usually defined in terms of the confidentiality, integrity, and availability (CIA) **triad**. The CIA triad can be briefly summarized as: data should only be available to authorized users, data should be correct and up-to-date, and data should be promptly available when needed. If one of these tenets is broken, this is usually a security incident. The CIA triad applies directly to malicious access, alteration, or destruction of AI system training data. But it might be a bit more difficult to see how the CIA triad applies to an AI system issuing decisions or predictions, and ML attacks tend to blend traditional data privacy and computer security concerns in confusing ways. So, let's go over an example of each.

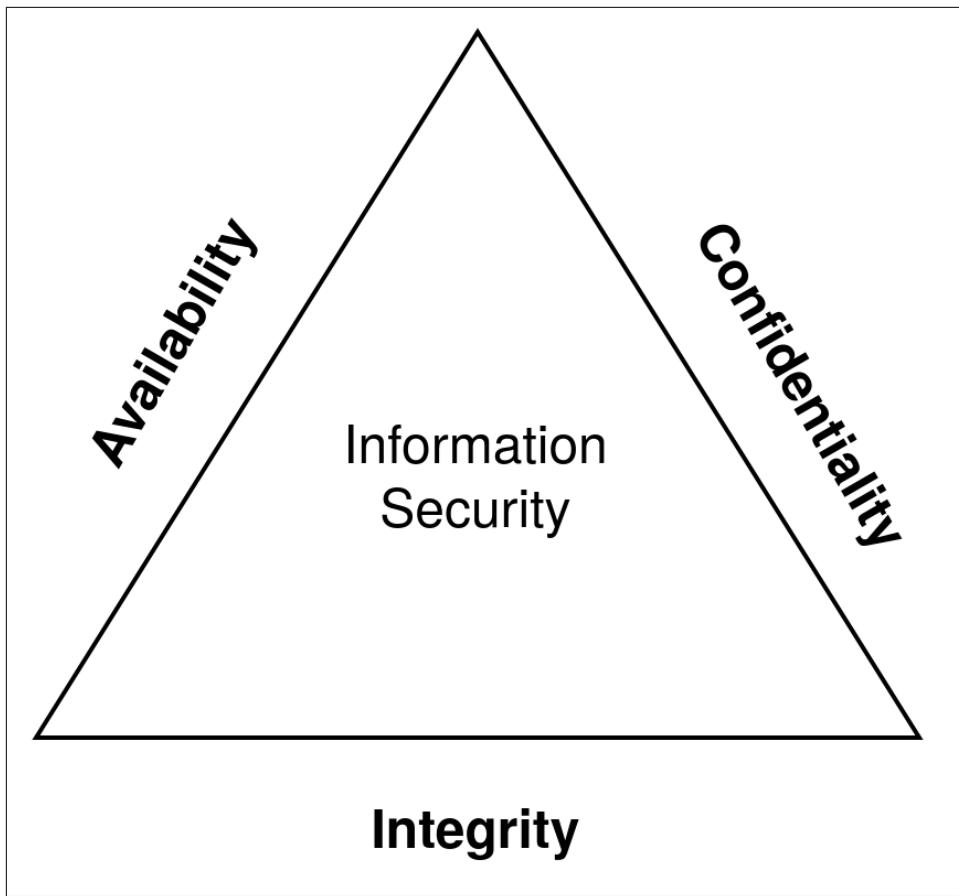


Figure 5-1. An illustration of the CIA triad for information security.

The confidentiality of an AI system can be breached by an inversion attack (see [inversion attack](#) below) in which a bad actor interacts with an application programming interface (API) in an appropriate manner, but uses explainable AI (XAI) techniques to extract information about your model and training data from their submitted input data and your system's predictions. In a more dangerous and sophisticated membership inference attack (see [membership inference attack](#) below), individual rows of training data, up to entire training datasets, can be extracted from AI system APIs or other endpoints. Note that these attacks can happen without unauthorized access to training files or databases, but result in the same security and privacy harms for your users or for your organization, including serious legal liabilities.

An AI system's integrity can be compromised by several means, such as data poisoning attacks or adversarial example attacks. In a data poisoning attack (see [data poisoning attack](#) below), an organizational insider subtly changes system training data to

alter system predictions in their favor. Only a small proportion of training data must be manipulated to change system outcomes, and specialized techniques from active learning and other fields can help attackers do so with greater efficiency. When AI systems apply millions of rules or parameters to thousands of interacting input features, it becomes nearly impossible to understand all the different predictions an AI system could make. In an adversarial example attack (see [adversarial example attack](#) below), an external attacker preys on such overly complex mechanisms by finding strange rows of data — adversarial examples — that evoke unexpected and improper outcomes from the AI system, and typically to benefit themselves at your expense.

The availability of an AI system is violated when users cannot get access to the services they expect. This can be a consequence of the attacks above bringing down the system, from more standard denial of service attacks, or from algorithmic discrimination. People depend on AI systems more and more in their daily lives, and when these models relate to high-impact decisions in government, finance, or employment, an AI system being down can deny users of essential services. Sadly, many AI systems also exhibit erroneous discrimination in outcomes and accuracy for historically marginalized demographic groups. Minorities may be less likely to experience the same levels of availability from automated credit offers or resume scanners. More directly frighteningly, they may be more likely to experience faulty predictions by facial recognition systems, including those used in security or law enforcement contexts.

These are just a few ways that an AI systems can experience security problems. There are many more. If you're starting to feel worried, keep reading! We'll discuss straightforward security concepts and best practices next. These tips can go a long way toward protecting any computer system.

Best Practices for Data Scientists

Starting with the basics will go a long way toward securing more complex AI systems. The following list summarizes those basics in the context of a data science workflow.

- **Access Control:** The less people that access sensitive resources the better. There are many sensitive components in an AI system, but locking down training data, training code, and deployment code to only those who require access will mitigate security risks related to data exfiltration, data poisoning, and backdoor attacks.
- **Bug Bounties:** Bug bounties, or when organizations offer monetary rewards to the public for finding vulnerabilities, are another practice from general information security that should probably also be applied to ML systems. Furthermore, audits, red teaming, and bug bounties need not be limited to security concerns alone. Bug bounties can be used to find all manner of problems in public-facing

AI systems, including algorithmic discrimination, unauthorized decisions, and product safety or negligence issues, in addition to security and privacy issues.

- **Incident Response Plans:** It's a common practice to have incident response plans in place for mission-critical IT infrastructure to quickly address any failures or attacks. Make sure those plans cover AI systems and have the necessary detail to be helpful if an AI system fails or suffers an attack. You'll need to nail down who does what when an AI incident occurs, especially in terms of business authority, technical know-how, budget and internal and external communications. There are excellent resources to help you get started with incident response from organizations like [NIST](#) and [SANS Institute](#). If you would like to see an example incident response plan for AI systems, checkout bnih.ai's [public resource page](#).
- **Routine Backups:** Ransomware attacks, where malicious hackers freeze access to an organization's IT systems — and delete precious resources if ransom payments are not made — are not uncommon. Make sure to backup important files on a frequent and routine basis to protect against both accidental and malicious data loss. It's also a best practice to keep physical backups unplugged (or "air-gapped") from any networked machines.
- **Least Privilege:** A strict application of the notion of least privilege, i.e., ensuring all personnel — even "rockstar" data scientists and ML engineers — receive the absolute minimum IT system permissions, is one of the best ways to guard against insider ML attacks. Pay special attention to limiting the number of root, admin, or super users.
- **Passwords and Authentication:** Use strong passwords, multi-factor authentication, and other authentication methods to ensure access controls and permissions are preserved. It's also not a bad idea to enforce a higher level of password hygiene, such as the use of password managers, for any personnel assigned to sensitive projects.
- **Physical Media:** Avoid the use of physical storage media for sensitive projects if at all possible, except when required for backups. Printed documents, thumb drives, backup media, and other portable data sources are often lost and misplaced by distracted data scientists and engineers. Worse still, they can be stolen by motivated adversaries. For less sensitive work, consider enacting policies and education around physical media use.
- **Product Security:** If your organization makes software, it's likely that they apply any number of security features and tests to these products. There's probably also no logical reason to not apply these same standards to public- or customer-facing AI systems. Reach out to security professionals in your organization to discuss applying standard product security measures to your AI systems.
- **Red Teams:** For mission-critical, or otherwise high stakes deployments of ML, systems should be tested under adversarial conditions. In what's known as "red

teaming,” as is done by [Facebook](#), teams of skilled practitioners attempt to attack AI systems and report their findings back to product owners. Like bug bounties, red-teaming can be highly-effective at finding security problems, but it’s not necessary to limit such audits only to security vulnerabilities.

- **Third Parties:** Building an AI system typically requires code, data, and personnel from outside your organization. Sadly, each new entrant to the build out increases your risk. Watch out for data poisoning in third party data or conducted by third party personnel. Scan all third-party packages and models for malware, and control all deployment code to prevent the insertion of backdoors or other malicious payloads.
- **Version and Environment Control:** To ensure basic security, you’ll need to know which changes were made to what files, when and by whom. In addition to version control of source code, any number of commercial or open source environment managers can automate tracking for large data science projects. Check out some of these open resources to get started with ML environment management: [dvc](#), [gigantum](#), [mlflow](#), [mlmd](#), and [modeldb](#).

ML security, to be discussed in the next sections, will likely be more interesting for data scientists than the mostly administrative and mundane ideas described here. However, because the security measures considered here are so simple, not following them could potentially result in legal liabilities for your organization, in addition to embarrassing or costly breaches and hacks. While still debated and somewhat amorphous, violations of security standards, as enforced by the [US Federal Trade Commission \(FTC\)](#) and other regulators, can bring with them unpleasant scrutiny and enforcement actions. Hardening the security of your AI systems is a lot of work, but failing to get the basics right can make big trouble when you’re building out more complex AI systems with lots of subsystems and dependencies.

Machine Learning Attacks

Various ML software artifacts, ML prediction APIs, and other AI system endpoints are now vectors for cyber and insider attacks. Such ML attacks can negate all the other hard work a data science team puts into mitigating other risks — because once your AI system is attacked, it’s not your system anymore. And attackers typically have their own agendas regarding accuracy, discrimination, privacy, stability, or unauthorized decisions. The first step to defending against these attacks is to understand them. We’ll go over an overview of the most well-known ML attacks below.

Integrity Attacks: Manipulated Machine Learning Outputs

Our tour of ML attacks will begin with attacks on ML model integrity, i.e., attacks that alter system outputs. Probably the most well-known type of attack, an adversarial

example attack, will be discussed first, followed by backdoor, data poisoning, and impersonation and evasion attacks. When thinking through these attacks, remember that they can often be used in two primary ways: (1) to grant attackers the ML outcome they desire, or (2) to deny a third-party their rightful outcome.

Adversarial Example Attacks

A motivated attacker can learn, by trial and error with a prediction API (i.e., “exploration” or “sensitivity analysis”), an **inversion attack**, or by social engineering, how to game your ML model to receive their desired prediction outcome or how to change someone else’s outcome. Carrying out an attack by specifically engineering a row of data for such purposes is referred to as an adversarial example attack. An attacker could use an adversarial example attack to grant themselves a loan, a lower than appropriate insurance premium, or to avoid pretrial detention based on a criminal risk score. See below for an illustration of a fictitious attacker executing an **adversarial example attack** on a credit lending model using strange rows of data.

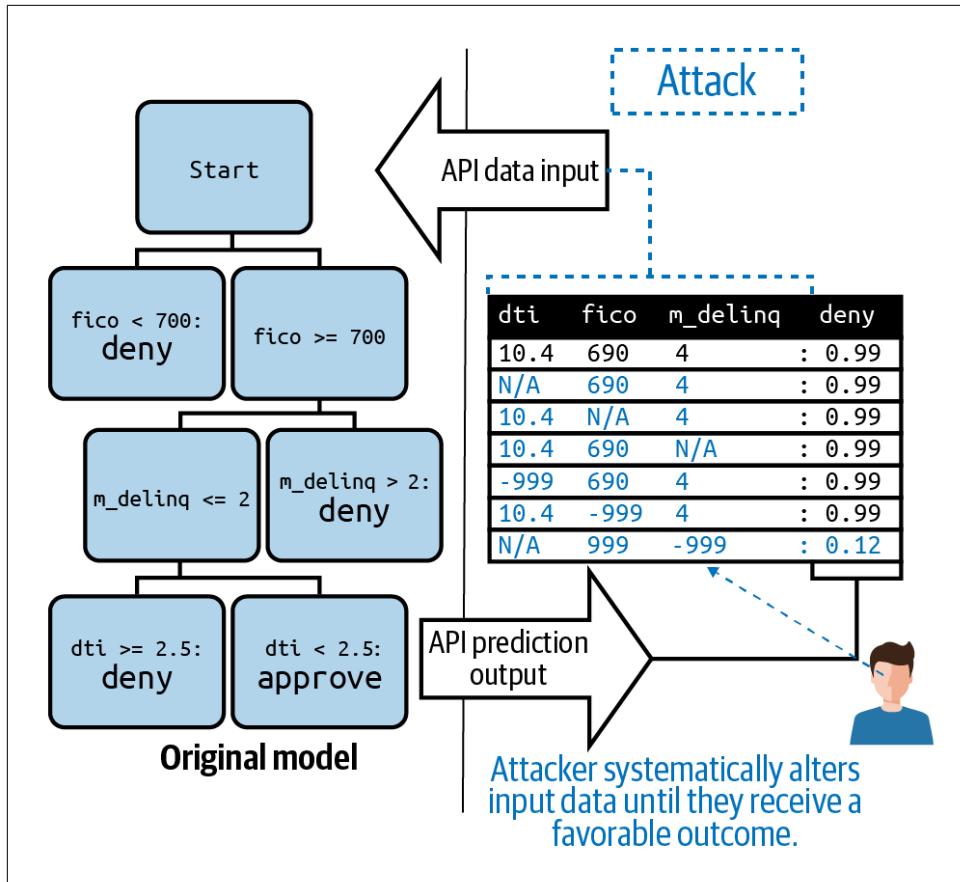


Figure 5-2. An illustration of an adversarial example attack. Adapted from the [Machine Learning Attack Cheatsheet](#).

Backdoor Attacks

Consider a scenario where an employee, consultant, contractor, or malicious external actor has access to your model's production code — code that makes real-time predictions. This individual could change that code to recognize a strange or unlikely combination of input variable values to trigger a desired prediction outcome. Like other outcome manipulation hacks, backdoor attacks can be used to trigger model outputs that an attacker wants, or outcomes a third party does not want. As depicted in the **back door attack** illustration, an attacker could insert malicious code into your model's production scoring engine that recognizes the combination of a realistic age but negative years on a job (*yoj*) to trigger an inappropriate positive prediction outcome for themselves or their associates. To alter a third-party's outcome, an attacker

could insert an artificial rule into your model's scoring code that prevents your model from producing positive outcomes for a certain group of people.

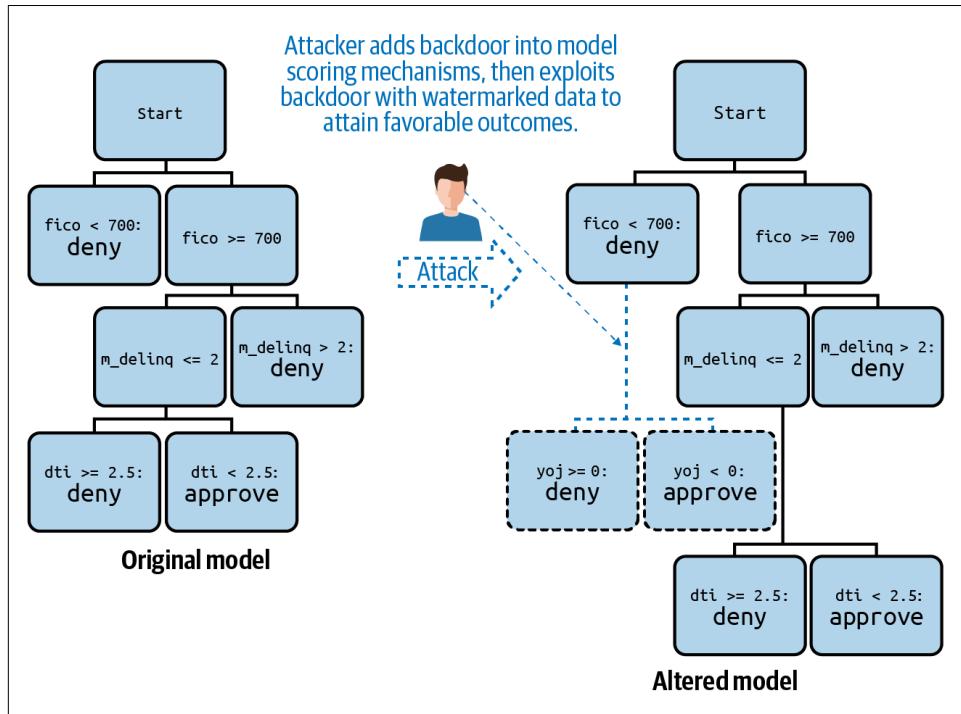


Figure 5-3. An illustration of a backdoor attack. Adapted from the [Machine Learning Attack Cheatsheet](#).

Data Poisoning Attacks

Data poisoning refers to someone systematically changing your training data to manipulate your model's predictions. To poison data, an attacker must have access to some or all of your training data. And at many companies, many different employees, consultants, and contractors have just that — and with little oversight. It's also possible a malicious external actor could acquire unauthorized access to some or all of your training data and poison it. A very direct kind of data poisoning attack might involve altering the labels of a training data set. In the [data poisoning attack](#) figure here, the attacker changes a small number of training data labels so that people with their kind of credit history will erroneously receive a credit product. It's also possible that a malicious actor could use data poisoning to train your model to intentionally discriminate against a group of people, depriving them the big loan, big discount, or low insurance premiums they rightfully deserve.

While it's simplest to think of data poisoning as changing the values in the existing rows of a data set, data poisoning can also be conducted by adding seemingly harmless or superfluous columns onto a data set and ML model. Altered values in these columns could then trigger altered model predictions. This is one of many reasons to avoid dumping massive numbers of columns into a black-box ML model.

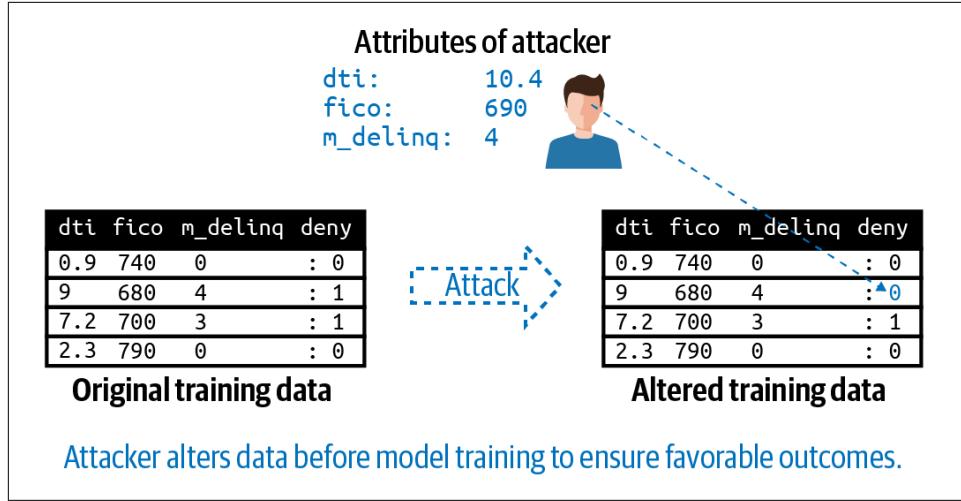


Figure 5-4. An illustration of a data poisoning attack. Adapted from the [Machine Learning Attack Cheatsheet](#).

Impersonation and Evasion Attacks

Using trial and error, a model **inversion attack**, or social engineering, an attacker can learn the types of individuals that receive a desired prediction outcome from your AI system. The attacker can then impersonate this kind of input or individual to receive a desired prediction outcome, or to evade an undesired outcome. These kinds of impersonation and evasion attacks resemble identity theft from the ML model's perspective. They're also similar to **adversarial example attacks**.

Like an adversarial example attack, an impersonation attack involves artificially changing the input data values to your model. Unlike an adversarial example attack, where a potentially random-looking combination of input data values could be used to trick your model, impersonation implies using the information associated with another modeled entity (i.e., customer, employee, financial transaction, patient, product, etc.) to receive the prediction your model associates with that type of entity. And evasion implies the converse — changing your own data to avoid an adverse prediction.

In the **impersonation** illustration below, an attacker learns what characteristics your model associates with awarding a credit product, and then falsifies their own infor-

mation to receive the credit. They could share their strategy with others, potentially leading to large losses for your company. Sound like science fiction? It's not. Closely related evasion attacks have worked for **facial-recognition payment and security systems**, and the **case** at the end of the Chapter will address several documented instances of evading ML security systems.

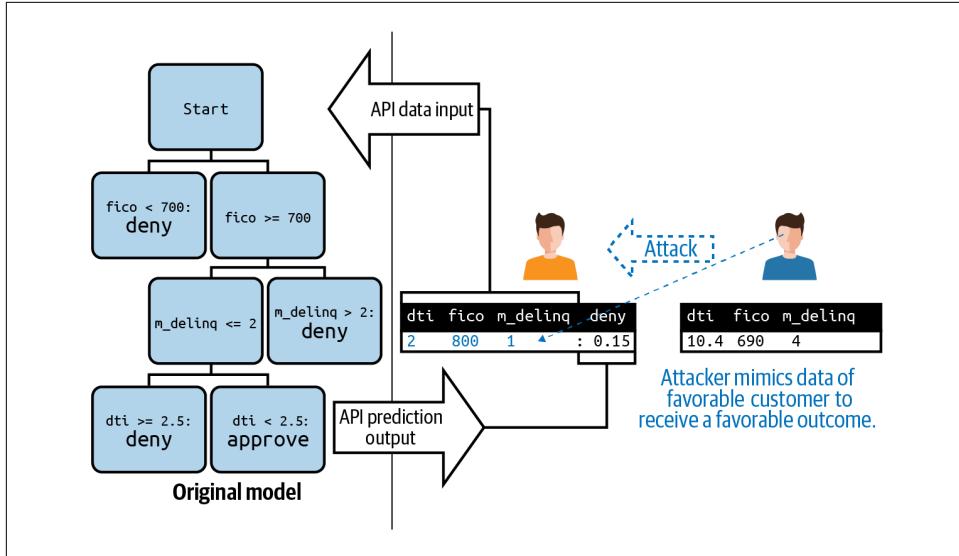


Figure 5-5. An illustration of an impersonation attack. Adapted from the *Machine Learning Attack Cheatsheet*.

Attacks on Machine Learning Explanations

In what has been called a “scaffolding” attack (see *Fooling LIME and SHAP: Adversarial Attacks on Post hoc Explanation Methods*), adversaries can poison post-hoc explanation like local interpretable model-agnostic explanations (LIME) and Shapley additive explanations (SHAP). Attacks on partial dependence, another common post-hoc explanation technique, have also been published recently (see *Fooling Partial Dependence via Data Poisoning*). Attacks on explanations can be used to alter both operator and consumer perceptions of an AI system. For example, to make another hack in the pipeline harder to find, or to make a discriminatory model appear fair (known as **fairwashing**). These attacks make clear that as ML pipelines and AI systems become more complex, bad actors could look to many different parts of the system, from training data all the way to post-hoc explanations, to alter system outputs.

Confidentiality Attacks: Extracted Information

Without proper countermeasures, bad actors can access sensitive information about your model and data. Model extraction and inversion attacks refer to hackers rebuild-

ing your model and extracting information from their copy of the model. Membership inference attacks allow bad actors to know what rows of data are in your training data, and even to rebuild your training data. Both attacks only require access to an unguarded AI system prediction API or other system endpoints.

Model Extraction and Inversion Attacks

Inversion basically means getting unauthorized information out of your model—as opposed to the normal usage pattern of putting information into your model. If an attacker can receive many predictions from your model API or other endpoint (website, app, etc.), they can train a surrogate model between their inputs and your system’s predictions. That extracted surrogate model is trained between the inputs the attacker used to generate the received predictions and the received predictions themselves. Depending on the number of predictions the attacker can receive, the surrogate model could become quite an accurate simulation of your model. Unfortunately, once the surrogate model is trained, you have several big problems:

- Models are really just compressed versions of training data. With the surrogate model, an attacker can start learning about your potentially sensitive training data.
- Models are valuable intellectual property. Attackers can now sell access to their copy of your model and cut into your return on investment.
- The attacker now has a sandbox from which to plan impersonation, adversarial example, membership inference, or other attacks against your model.

Such surrogate models can also be trained using external data sources that can be somehow matched to your predictions, as ProPublica [famously did](#) with the proprietary COMPAS criminal risk assessment instrument.

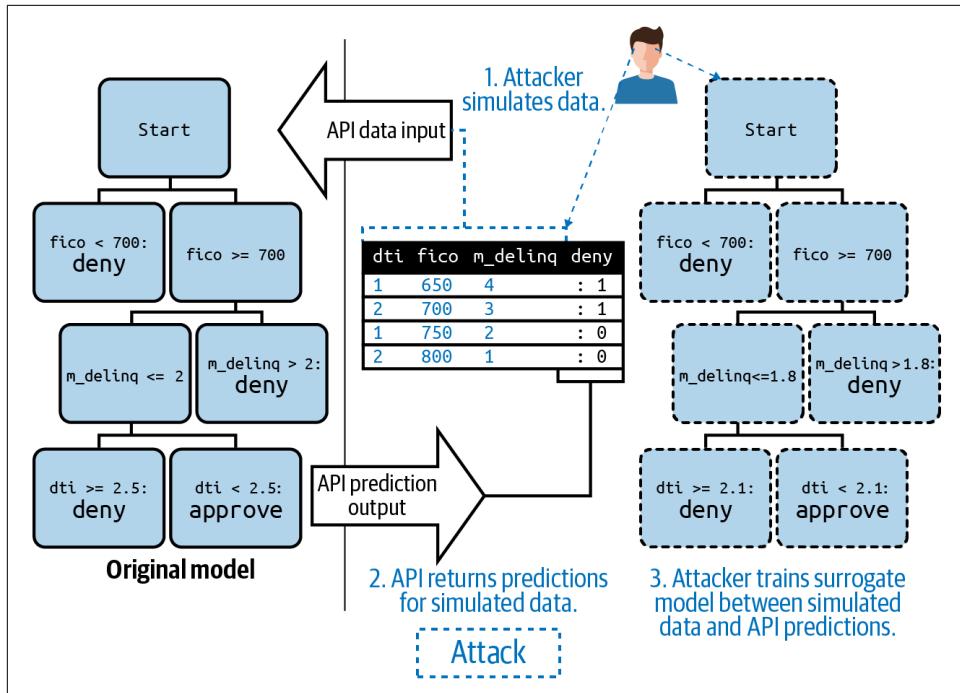


Figure 5-6. An illustration of an inversion attack. Adapted from the [Machine Learning Attack Cheatsheet](#).

Membership Inference Attacks

In an attack that starts with model extraction, and also carried out by surrogate models, a malicious actor can determine whether a given person or product is in your model's training data. Called a **membership inference** attack, this hack is executed with two layers of surrogate models. First an attacker passes data into a public prediction API or other endpoint, receives predictions back, and trains a surrogate model or models between the passed data and the predictions. Once a surrogate model or models has been trained to replicate your model, the attacker then trains a second layer classifier that can differentiate between data that was used to train the first surrogate model and data that was not used to train that surrogate. When this second model is used to attack your model, it can give a solid indication as to whether any given row (or rows) of data is in your training data.

Membership in a training data set can be sensitive when the model and data are related to undesirable outcomes such as bankruptcy or disease, or desirable outcomes like high income or net worth. Moreover, if the relationship between a single row and the target of your model can be easily generalized by an attacker, such as an obvious relationship between race, gender, age and some undesirable outcome, this attack can violate the privacy of an entire class of people. Frighteningly, when carried out to its

fullest extent, a membership inference attack could also allow a malicious actor, with access only to an unprotected public prediction API or other model endpoint, to reconstruct portions of a sensitive or valuable training data set.

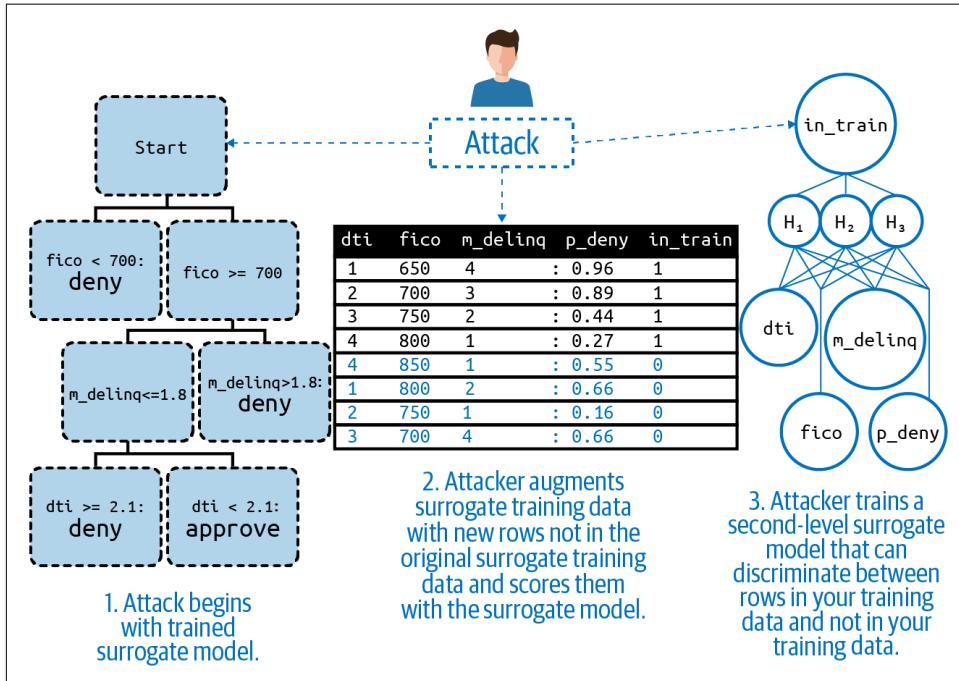


Figure 5-7. An illustration of a membership inference attack. Adapted from the [Machine Learning Attack Cheatsheet](#).

While the attacks discussed above are some of the most well-known types of attacks, keep in mind that these are not the only types of ML hacks, and that new attacks can emerge very quickly. Accordingly, we'll also address a few general concerns to help you frame the broader threat environment, before moving on to countermeasures you can use to protect your AI system.

General AI Security Concerns

One common theme of this book is that AI systems are fundamentally software systems, and applying common sense software best practices to AI systems is usually a good idea. The same applies for security. As software systems and services, AI systems exhibit similar failure modes, and experience the same attacks, as general software systems. What are some of these general concerns? Unpleasant things like intentional abuses of AI technology, availability attacks, trojans and malware, man-

in-the-middle attacks, unnecessarily complex black-box systems, and the woes of distributed computing.

- **Abuses of Machine Learning:** Nearly all tools can also be weapons, and ML models and AI systems can be abused in numerous ways. Let's start by considering deep fakes. Deep fakes are an application of deep learning that can, when done carefully, seamlessly blend fragments of audio and video into convincing new media. While deep fakes can be used to bring movie actors back to life, as was done in some recent Star Wars films, deep fakes can be used to harm and extort people. Of course non-consensual pornography, in which the victim's face is blended into an adult video, is one of the most popular uses of deep fakes, as documented by the BBC and other news outlets. Deep fakes have also been implicated in financial crimes, e.g., when an attacker [used a CEO's voice](#) to order money transferred into their own account. Algorithmic discrimination is another common application of abusive AI. In a "fairwashing" attack, post-hoc explanations can be altered to hide discrimination in a biased model. And facial recognition can be used directly for [racial profiling](#). We're touching on just a few of the ways AI systems can be abused, for a broader treatment of this important topic see [*AI-enabled future crime*](#).
- **Availability Attacks:** AI systems can fall victim to more general denial of service (DOS) attacks, just like other public-facing services. If a public-facing AI system is critical to your organization, make sure it's hardened with firewalls and filters, reverse domain name server system (DNS) lookup, and other countermeasures that increase availability during a DOS attack. Unfortunately, we also have to think through another kind of availability failure for AI systems — those caused by algorithmic discrimination. If algorithmic discrimination is severe enough, driven by internal failures or adversarial attacks, your AI system will likely not be usable by large portion of its users. Be sure to test for algorithmic discrimination during training and throughout a system's deployed life cycle.
- **Trojans and Malware:** Machine learning in the research and development environment is highly dependent on a diverse ecosystem of open source software packages. Some of these packages have many, many contributors and users. Some are highly specific and only meaningful to a small number of researchers or practitioners. It's well understood that many packages are maintained by brilliant statisticians and machine learning researchers whose primary focus is mathematics or algorithms, not software engineering, and certainly not security. It's not uncommon for a machine learning pipeline to be dependent on dozens or even hundreds of external packages, any one of which could be hacked to conceal an attack payload. Third party packages with large binary data stores and pre-trained ML models seem especially ripe for these kinds of problems. If possible, scan all software artifacts associated with an AI systems for malware and trojans.

- **Man in the Middle Attacks:** Because many AI system predictions and decisions are transmitted over the internet or an organization's network, they can be manipulated by bad actors during that journey. Where possible, use encryption, certificates, mutual authentication, or other countermeasures to ensure the integrity of AI system results passed across networks.
- **Black-box Machine Learning:** Although recent developments in interpretable models and model explanations have provided the opportunity to use accurate and also transparent models, many machine learning workflows are still centered around black-box models. Such black-box models are a common type of, often unnecessary, complexity in a commercial ML workflow. A dedicated, motivated attacker can, over time, learn more about your overly complex black-box ML model than you or your team knows about it. (Especially in today's overheated and turnover-prone data science job market.) This knowledge imbalance can potentially be exploited to conduct the attacks described above or for other yet unknown types of attacks.
- **Distributed Computing:** For better or worse, we live in the age of big data. Many organizations are now using distributed data processing and AI systems. Distributed computing can provide a broad attack surface for a malicious internal or external actor. Data could be poisoned on only one or a few worker nodes of a large distributed data storage or processing system. A back door could be coded into just one model of a large ensemble. Instead of debugging one simple data set or model, now practitioners must sometimes examine data or models distributed across large computing clusters.

Starting to get worried again? Hang in there — we'll cover countermeasures for confidentiality, integrity, and availability attacks on AI systems next.

Counter-measures

There are many countermeasures you can use and, when paired with the processes proposed in Chapter 1, bug bounties, security audits, and red teaming, such measures are more likely to be effective. Additionally, there are the newer subdisciplines of adversarial ML and robust ML, that are giving the full academic treatment to these subjects. This section will outline some of the defensive measures you can use to help make your AI systems more secure, including model debugging for security, model monitoring for security, privacy enhancing technologies, robust machine learning, and a few general approaches.

Model Debugging for Security

ML models can and should be tested for security vulnerabilities before they are released. In these tests, the goal is basically to attack your own AI systems, to under-

stand your level of vulnerability, and to patch up any discovered vulnerabilities. Some general techniques, that work across different types of ML models, for security debugging are adversarial example searches and sensitivity analysis, audits for insider attacks and model extraction attacks, and discrimination testing.

Adversarial Example Searches and Sensitivity Analysis

Conducting sensitivity analysis with an adversarial mindset, or better yet, conducting your own adversarial example attacks, is a good way to determine if your system is vulnerable to perhaps the simplest and most common type of ML integrity attack. The idea of these ethical hacks is to understand what feature values (or combinations thereof) can cause large swings in your system's output predictions. If you're working in the deep learning space, packages like `cleverhans` and `foolbox` can help you get started with testing your AI system. For those working with structured data, good old sensitivity analysis can go a long way toward pointing out instabilities in your system. You can also use genetic learning to evolve your own adversarial examples or you can use **heuristic methods based on individual conditional expectation (ICE)** to find adversarial examples. Once you find instabilities in your AI system, triggered by these adversarial examples, you'll want to use cross-validation or regularization to train a more stable model, apply techniques from **robust machine learning**, or explicitly monitor for the discovered adversarial examples in real time. You should also link this information to the system's incident response plan, in case it's useful later.

Auditing for Insider Data Poisoning

If a data poisoning attack were to occur, system insiders — employees, contractors, and consultants — are not unlikely culprits. How can you track down insider data poisoning? First, score those individuals with your system. Any insider receiving a positive outcome could be the attacker or know the attacker. Because a smart attacker will likely perform the minimum changes to the training data that result in a positive outcome, you can also use residual analysis to look for beneficial outcomes with larger than expected residuals, indicating the ML model may have been inclined to issue a negative outcome for the individual had the training data not been altered. Data and environment management are strong countermeasures for insider data poisoning too, as any changes to data are tracked with ample metadata (who, what, when, etc.). You can also try the reject on negative impact (RONI) technique, proposed in the seminal *The Security of Machine Learning* paper, to remove potentially altered rows from system training data.

Discrimination Testing

DOS attacks, resulting from some kind of algorithmic discrimination — intentional or not — are a plausible type of availability attack. In fact, it's already happened. In 2016, Twitter users poisoned the `Tay Chatbot` to the point where only those users

interested in neo-nazi pornography would find the system's service appealing. This type of attack could also happen in a more serious context, such as employment, lending or medicine, where an attacker uses data poisoning, model backdoors, or other types of attacks to deny service to a certain group of customers. This is one of the many reasons to conduct discrimination testing, and remediate any discovered discrimination, both at training time and as part of regular model monitoring. There are several great open source tools for detecting discrimination and making attempts to remediate it, such as [aequitas](#), [Themis](#), and [AIF360](#).

Ethical Hacking: Model Extraction Attacks

Model extraction attacks are harmful on their own, but they are also the first stage for a membership inference attack. Conduct your own model extraction attacks to determine if your system is vulnerable to these confidentiality attacks. If you find some API or model end-point that allows you to train a surrogate model between input data and system outputs, lock it down with solid authentication and throttle any abnormal requests at this endpoint. Because a model extraction attack may have already happened via this endpoint, analyze your extracted surrogate models as follows:

- What are the accuracy bounds of different types of surrogate models? Try to understand the extent to which a surrogate model can really be used to gain knowledge about your AI system.
- What types of data trends can be learned from your surrogate model? Like linear trends represented by linear model coefficients or course summaries of population subgroups in a surrogate decision tree.
- What rules can be learned from a surrogate decision tree? For example, how to reliably impersonate an individual who would receive a beneficial prediction? Or how to construct effective adversarial examples?

If you see that it is possible to train an accurate surrogate model from one of your system endpoints, and to answer some of the questions above, then you'll need to take some next steps. First, conduct a membership inference attack on yourself to see if that two stage attack would also be possible. You'll also need to record all of the information related to this ethical hacking analysis and link to it to the system's incident response plan. Incident responders may find this information helpful at a later date, and it may unfortunately need to be reported as a breach if there is strong evidence that an attack has occurred.

Debugging security vulnerabilities in your AI system is important work that can save you future money, time and heartache, but so is watching your system to ensure it stays secure. Next we'll take up model monitoring for security.

Model Monitoring For Security

Once hackers can manipulate or extract your ML model, it's really not your model anymore. To guard against attacks on your ML, you'll not only need to train and debug it with security in mind, you'll also need to monitor it closely once it goes live. Monitoring for security should be geared toward algorithmic discrimination, anomalies in input data queues, anomalies in predictions, and high-usage. Here are some tips on what and how to monitor:

- **Discrimination monitoring:** Discrimination testing, as discussed in previous chapters, must be applied during model training. But for many reasons, including unintended consequences and malicious hacking, discrimination testing must be performed during deployment too. If discrimination is found during deployment, it should be investigated and remediated. This helps to ensure a model that was fair during training remains fair in production.
- **Input anomalies:** Unrealistic combinations of data, that could be used to trigger back doors in model mechanisms, should not be allowed into model scoring queues. Anomaly detection ML techniques, like autoencoders and isolation forests, may be generally helpful in tracking problematic input data. However, you can also use commonsense data integrity constraints to catch problematic data before it hits your model. An example of such unrealistic data is reporting an age of 40 years and a job tenure of 50 years. If possible, you should also consider monitoring for random data, training data, or duplicate data. Because random data is often used in model extraction and inversion attacks, build out alerts or controls that help your team understand if or when your model may be encountering batches of random data. Real-time scoring of rows that are extremely similar or identical to data used in training, validation, or testing should be recorded and investigated as they could indicate a membership inference attack. Finally, be on the lookout for duplicate data in real-time scoring queues, as this could indicate an evasion or impersonation attack.
- **Output anomalies:** Output anomalies can be indicative of adversarial example attacks. When scoring new data, compare your ML model prediction against a trusted, transparent benchmark model or a benchmark model trained on a trusted data source and pipeline. If the difference between your more complex and opaque ML model and your interpretable or trusted model is too great, fall back to the predictions of the conservative model or send the row of data for manual processing. Statistical control limits, which are akin to moving confidence intervals, can also be used to monitor for anomalous outputs.
- **Meta-monitoring:** Monitor the basic operating statistics — the number of predictions in some time period, latency, CPU, memory, and disk loads, or the number of concurrent users — to ensure your system is functioning normally. You can even train an autoencoder-based anomaly detection metamodel on your

entire AI system's operating statistics and then monitor this metamodel for anomalies. An anomaly in system operations could tip you off that something is generally amiss in your AI system.

Monitoring for attacks is one the most pro-active steps you can take to counter ML hacks. However, there are still a few more countermeasures to discuss. We'll look into privacy enhancing technologies (PETs) below.

Privacy-enhancing Technologies

Privacy-preserving ML is a research subdiscipline with direct ramifications for the confidentiality of your ML training data. While just beginning to gain steam in the ML and ML operations (MLOps) communities, PETs can give you an edge when it comes to protecting your data. Some of the most promising and practical techniques from this emergent field include federated learning and differential privacy.

Federated learning

Federated learning is an approach to training ML models across multiple decentralized devices or servers holding local data samples, without exchanging raw data between them. This approach is different from traditional centralized ML techniques where all datasets are uploaded to a single server. The main benefit of federated learning is that it enables the construction of ML models without sharing data among many parties. Federated learning avoids sharing data by training local models on local data samples and exchanging parameters between servers or edge devices to generate a global model, which is then shared by all servers or edge devices. Assuming a secure aggregation process is used, federated learning helps address fundamental data privacy and data security concerns. Among other open source resources, look into [PySyft](#) or [FATE](#) to start learning about implementing federated learning at your organization (or with partner organizations!).

Differential Privacy

Differential privacy is a system for sharing information about a dataset by describing patterns about groups in the dataset without disclosing information about specific individuals. In ML tools, this is often accomplished using specialized types of differentially private learning algorithms. This makes it more difficult to extract sensitive information from training data or a trained ML model in model extraction, model inversion or membership inference attacks. In fact, an ML model is said to be differentially private if an outside observer cannot tell if an individual's information was used to train the model. There are lots of high-quality open source repositories to check out and try, including:

- Google's [differential-privacy](#)

- IBM's `diffprivlib`
- Tensorflow's `privacy`

Many ML approaches that invoke differential privacy are based on **differentially private stochastic gradient descent (DP-SGD)**. DP-SGD injects structured noise into gradients determined by SGD at each training iteration. In general, DP-SGD and related techniques ensure that ML models do not memorize too much specific information about training data. Because they prevent ML algorithms from focusing on particular individuals, they could also lead to increased generalization performance and fairness benefits.

You may hear about confidential computing or homomorphic encryption under the PET topic heading as well. These are promising research and technology directions to watch for the future. Another subdiscipline of ML research to watch is robust ML, which can help you counter adversarial example attacks and other adversarial manipulation of your AI system.

What else do data scientists need to know about privacy?

Data is becoming a more regulated quantity, even in the U.S. Aside from regulation, being sloppy with data can harm our users, our organization, or the general public. As data scientists we need to know some data privacy basics:

- **Regulations and policy:** We are likely to be operating under some data privacy regulation(s) or organizational privacy policy. We should know the basics of what our obligations and try to adhere to them. In the U.S., healthcare and educational data are particularly sensitive, but the E.U. General Data Protection Regulation (GDPR) — to which many multinationl organizations must comply — covers nearly all types of consumer data and U.S. states are passing many new data privacy laws.
- **Consent:** Although far from perfect, many data privacy regulations rely on the notion of *consent*, i.e., that consumers actively opt-in to the use of their data for a specific applications. Both from a legal and ethical perspective, it's good to confirm that we have consent to use our data for the model we're training.
- **Other legal bases for use:** If we are operating under the GDPR we need a legal basis to use much of the consumer data we might be interested in for ML. Consent is often the legal basis, but there are other examples: contractual obligations, governmental tasks, or medical emergencies.
- **Anonymization:** It's almost never a good idea to work with personal identifiable information (PII). Data like social security numbers, phone numbers, and even email addresses — known as *direct identifiers* — allows bad actors to tie private and sensitive information back to specific people. Even combinations of demographic information, like age, race, or gender — known as *indirect identifiers* —

can be used to tie information back to specific individuals. Generally, this kind of data should be removed, masked, hashed, or otherwise anonymized for use in model training, for privacy and bias reasons.

- **Biometric data:** Data like digital images, videos, fingerprints, voiceprints, iris scans, genomic data, or other data that encodes biometric information typically requires additional security and data privacy controls.
- **Retention limits or requirements:** Laws and organizational privacy policies may define retention limits — or how long data can be saved before mandatory deletion. (We've seen two-week retention limits!) We may also have to deal with legal or organizational retention requirements that enforce how long data must be stored and kept private and secure — this can be for years.
- **Deletion and rectification requests:** Many laws and policies enable consumers to have their data updated, corrected or fully deleted upon request, and this could potentially affect ML training data.
- **Explanation:** Many data privacy laws also seem to be putting forward a requirement for explaining automatic processing of data that affects consumers. Although vague and largely unsettled, this may mean that in the future many more ML decisions will have to be explained to consumers.
- **Intervenability:** Similar to explanation, many new laws also seem to enstate a requirement for *intervenability*, which is similar to the appeal and override concepts of actionable recourse.

All of these issues can have a serious impact on our ML workflows. We may need to plan to deal with retention limits, deletion requests, rectification requests, or intervenability requirements. For example, would we need to decommission and delete a model trained on expired data or data consumers request to delete? No one is quite sure yet, but it's not impossible. If you've never received training about your data privacy obligations, or have questions or concerns about data privacy topics, it might not be a bad idea to check with your manager or legal department.

Robust Machine Learning

Robust ML includes many cutting-edge ML algorithms developed to counter adversarial example attacks, and to a certain extent data poisoning as well. The study of robust ML gained momentum after several researchers showed that silly, or even invisible, changes to input data can result in huge swings in output predictions for computer vision systems. Such swings in model outcomes are a troubling sign in any domain, but when considering medical imaging or semi-autonomous vehicles, they are downright dangerous. Robust ML models help enforce stability in model outcomes, and strikingly, an important notion from fairness, that similar individuals be treated similarly. Similar individuals in ML training data or live data are individuals that are close to one another in the Euclidean space of the data. Robust ML tech-

niques often try to establish a hypersphere around individual examples of data, and ensure that other similar data within the hypersphere receive similar predictions. Whether caused by bad actors, over-fitting, underspecification, or other factors, Robust ML approaches help protect your organization from risks arising from unexpected predictions. Interesting papers and code are hosted at the [Robust Machine Learning](#) site, and Madry group at Massachusetts Institute of Technology (MIT) has even published a full [Python package](#) for robust ML.

General Countermeasures

There are a number of catchall countermeasures that can defend against several different types of ML attacks, including authentication, throttling, and watermarking. Many of these same kinds of countermeasures are also best practices for AI systems in general, like interpretable models, model management, and model monitoring. The last topic before the case study is a brief description of important and general countermeasures against AI system attacks.

- **Authentication:** Whenever possible, disallow anonymous use for high-stakes AI systems. Login credentials, multi-factor authentication, or other types of authentication that force users prove their identity, authorization, and permission to use a system put a blockade between your model API and bad actors.
- **Interpretable, fair, or private models:** Modeling techniques now exist (e.g., monotonic GBMs (M-GBM), [scalable Bayesian rule lists \(SBRL\)](#), [eXplainable Neural Networks \(XNN\)](#)), that can allow for both accuracy and interpretability in ML models. These accurate and interpretable models are easier to document and debug than classic ML black boxes. Newer types of fair and private modeling techniques (e.g., [LFR](#), [DP-SGD](#)) can also be trained to downplay outward visible, demographic characteristics that can be observed, socially engineered into an adversarial example attack, or impersonated. These models, enhanced for interpretability, fairness or privacy, should be more easily debugged, more robust to changes in an individual entity's characteristics, and more secure than over-used ML black boxes.
- **Model Documentation:** Model documentation is a risk-mitigation strategy that has been used for decades in banking. It allows knowledge about complex modeling systems to be preserved and transferred as teams of model owners change over time, and for knowledge to be standardized for efficient analysis by model validators and auditors. Model documentation should cover the who, when, where, what and how of an AI system, including many details from contact information for stakeholders to algorithmic specification. Model documentation is also a natural place to record any known vulnerabilities or security concerns for an AI system, enabling future maintainers or other operators that interact with

the system to allocate oversight and security resources efficiently. Incident response plans should also be linked to model documentation.

- **Model Management:** Model management typically refers to a combination of process controls, like documentation, combined with technology controls, like model monitoring and model inventories. Organizations should have an exact count of deployed AI systems, and inventory associated code, data, documentation and incident response plans in a structured manner, and monitor all deployed models. These practices make it easier to understand when something goes wrong and to deal with problems quickly when they arise.
- **Throttling:** When high use or other anomalies, such as adversarial examples, or duplicate, random, or training data, are identified by model monitoring systems, consider throttling prediction APIs or other system endpoints. Throttling can refer to restricting high numbers of rapid predictions from single users, artificially increasing prediction latency for all users, or other methods that can slow down attackers conducting model or data extraction attacks and adversarial example attacks.
- **Watermarking:** Watermarking refers to adding a subtle marker to your data or predictions, for the purpose of deterring theft of data or models. If data or predictions carry identifiable traits, such as actual watermarks on images or sentinel markers in structured data, it can make stolen assets harder to use and more identifiable to law enforcement or other investigators once a theft occurs.

Applying these general defenses and best practices, along with some of the more specific countermeasures discussed in previous sections, is a great way to achieve a high level of security for an AI system. And now that you've covered security basics, ML attacks, and many countermeasures for those attacks, you're armed with the knowledge you need to start red-teaming your organization's AI --- especially if you can work with your organization's IT security professionals. We'll now examine some real-world AI security incidents to provide additional motivation for doing the hard work of red-teaming AI, and to gain insights into some of today's most common AI security issues.

Case Study: Real-world Evasion Attacks

AI systems used for both physical and online security have suffered evasion attacks in recent years. Below, the case discussion touches on evasion attacks used to avoid Facebook filters and perpetuate disinformation and terrorist propaganda, and evasion attacks against real-world payment and physical security systems.

As the COVID pandemic ground on and the 2020 US presidential campaign was in high gear, those proliferating disinformation related to both topics took advantage of weaknesses in Facebook's manual and automated content filtering. As reported by

NPR, [Tiny Changes Let False Claims About COVID-19, Voting Evade Facebook Fact Checks](#). While Facebook uses news organizations such as Reuters and the Associated Press to fact-check claims made by its billions of users, it also uses AI-based content filtering, particularly to catch copies of human-identified misinformation posts. Unfortunately, minor changes, as simple as different backgrounds or fonts, image cropping, or simply describing memes with words instead of images, allowed bad actors to circumvent Facebook's AI fact-checking filters. In their defense, Facebook does carry out enforcement actions against many offenders, including limiting the distribution of posts, not recommending posts or groups, and demonetization. Yet, according to one advocacy group, Facebook fails to catch about [42% of disinformation posts containing information flagged by human fact checkers](#). The same advocacy group — Avaaz — estimates that a sample of just 738 unlabeled disinformation posts lead to an estimated 142 million views and 5.6 million user interactions.

Recent events have shown us that online disinformation and security threats can spill over into the real world. Disinformation about the 2020 US election and the COVID pandemic are thought to be primary drivers of the frightening events of the January 6th, 2021 US Capital riots. In perhaps even more frightening evasion attacks, the BBC has reported that [ISIS operatives continue to evade Facebook content filters](#). By blurring logos, splicing their videos with mainstream news content, or just using strange punctuation, ISIS members or affiliates have been able to post propaganda, explosive-making tutorials, and even evasion attack tutorials to Facebook, garnering tens of thousands views for their violent, disturbing, and vitriolic content. While evasion attacks on AI-based filters are certainly a major culprit, there are also less human moderators for Arabic content on Facebook. Regardless of whether its humans or machines failing at the job, this type of content can be truly dangerous, contributing both to radicalization and real-world violence. Physical evasion attacks are also a specter for the near future. Researchers recently showed that [some AI-based physical security systems are easy targets for evasion attacks](#). With permission of system operators, researchers used lifelike 3-dimensional masks to bypass facial recognition security checks on AliPay and WeChat payment systems. In one egregious case, researchers were even able to use a picture of another person on an iPhone screen to board a plane at Amsterdam's Schiphol Airport.

Lessons Learned

Taken together, bad actors' evasion of online safeguards to post dangerous content, and evasion of physical security systems to make monetary payments and to travel by plane paints a scary picture of a world where AI security is not taken seriously. What lessons learned from Chapter 5 could be applied to prevent these evasion attacks? The first lesson is related to robust ML. AI systems used for high-stakes security applications, be it online or real-world, must not be fooled by tiny changes to normal system inputs. Robust ML, and related technologies, must progress to the point where sim-

plastic evasion techniques, like blurring of logos or changes to punctuation, are not effective evasion measures. Another lesson comes from the beginning of the Chapter: the adversarial mindset. Anyone who thought seriously about security risks for these AI-based security systems should have realized masks, or just other images, were an obvious evasion technique. Thankfully, it turns out that some organizations do employ countermeasures for adversarial scenarios. Better facial recognition security systems deploy techniques meant to ensure the liveness of the subjects they are identifying. The better facial recognition systems also employ discrimination testing to ensure availability is high, and error rates are as low as possible, for all their users.

Another major lesson to be learned from real-world evasion attacks pertains to the responsible use of technology in general, and AI in particular. Social media has proliferated beyond physical borders and its complexity has grown past many country's abilities to effectively regulate it. With a lack of government regulation, users are counting on social media companies to regulate themselves. Being tech companies, social networks often rely on more technology, like AI-based content filters, to retain control of their systems. But what if those controls don't really work? As technology and AI play larger roles in human lives, lack of rigor and responsibility in their design, implementation, and deployment will have ever-increasing consequences. Those designing technologies for security or other high-stakes applications have an especially serious obligation to be realistic about today's AI capabilities and apply process and technology controls to ensure adequate real-world performance. This chapter laid out how to red-team AI systems, and apply process and technology controls, for important security vulnerabilities. The next Chapter will introduce a wide array techniques to test and ensure real-world performance beyond security.

Resources

- *A Marauder's Map of Security and Privacy in Machine Learning*
- BIML Interactive Machine Learning Risk Framework
- FTC Start With Security Guidelines
- Mitre Adversarial Threat Matrix
- NIST Computer Security Resource Center

Explainable Boosting Machines and Explaining XGBoost

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Chapter 6 explores interpretable models and post-hoc explanation with [examples](#) relating to consumer finance. It also applies the approaches discussed in Chapter 2 using explainable boosting machines (EBMs), monotonically constrained XGBoost models, and post-hoc explanation techniques. We’ll start with a concept refresher for additivity, constraints, partial dependence and individual conditional expectation (ICE), and model documentation.

We’ll then explore an example credit underwriting problem by building from a penalized regression, to a generalized additive model (GAM), to an EBM. In working from simpler to more complex models, we’ll document explicit and deliberate trade-offs regarding the introduction of nonlinearity and interactions into our example probability of default classifier, all while preserving near total explainability with additive models.

After that, we'll consider a second approach to predicting default that allows for complex feature interactions, but controls complexity with monotonic constraints based in causal knowledge. Because the monotonically constrained gradient boosting machine (GBM) won't be interpretable on its own, we'll pair it with robust post-hoc explanation techniques for greatly enhanced explainability. Finally, Chapter 6 will close with a discussion of the pros and cons of popular Shapley value methods.

Concept Refresher: ML Transparency

Before diving into technical examples, let's review some of the key concepts Chapter 2. Because our first example will highlight the strengths of the GAM family of models, we'll address additivity below, particularly in comparison to models that allow for high-degree interactions. Our second example will use monotonicity constraints to affect a poor man's causality approach with XGBoost, so we'll briefly highlight the connections between causation and constraints below. We'll also be using partial dependence and ICE to compare and assess our different approach's treatment of input features, so we'll need a quick refresh on the strengths and weaknesses of those post-hoc explainers, and we'll need to go over the importance of model documentation one more time — because model documentation is that important!

Additivity vs. Interactions

A major distinguishing characteristic of black-box machine learning (ML) is its propensity to create extremely high-degree interactions between input features. It's thought that this ability to consider the values of many features in simultaneous combination increases the predictive capacity of ML models versus more traditional linear or additive models that tend to consider input features independently. But it's been shown that black-box models are **not more accurate for structured data**, like credit underwriting data, and all those interactions in black-box models are very difficult for people to understand. Moreover, high-degree interactions also lead to instability, because small changes in one or few features can interact with other features to dramatically change model outcomes, and high-degree interactions lead to overfitting, because today's relevant seventeen-way interactions are likely not tomorrow's relevant seventeen-way interactions.

Another important characteristic of ML is the ability to learn non-linear phenomenon in training data, *automatically*. It turns out that if we can separate nonlinearity from interactions, we can realize substantial boosts in predictive quality, while preserving a great deal of explainability, if not complete explainability. This is the magic of GAMs! And EBMs are the next step that allow us to introduce a sane number of two-way interactions, in an additive fashion, that can result even better performance. Later in this chapter, the **GAM family** example aims to provide an object lesson in these trade-offs, starting with a straightforward, additive, linear model baseline, then

introducing nonlinearity with GAMs, and finally introducing understandable two-way interactions with EBMs. When we introduce nonlinearity and interactions carefully, as opposed to assuming more complexity is always better, it enables us to justify our modeling approach, tether it to real-world performance concerns, and to generate a lot of interesting plots of feature behavior. These justifications and plots are also great materials for subsequent model documentation, to be addressed in greater detail below.

Steps Toward Causality with Constraints

Causal discovery and inference are important directions in the future of predictive modeling. Why? Because when we build on correlation with ML, we are building on sand. Correlations change constantly in the real-world and can be spurious or just wrong. If we can base models on causal relationships instead or just memorializing some snapshot of complex correlations with an ML model, we greatly decrease overfitting, data drift, and social bias risks. As of now, causal methods tend to be somewhat difficult for most organizations to implement, so our [monotonicity constraints](#) example highlights a simple and easy step we can take to inject causality into ML models. If we can use our brains, or simple but robust experiments, to understand the directionality of a causal relationship in the real world, we can use monotonicity constraints to enforce that directionality in our XGBoost models. For instance, if I *know* that an increasing number of late payments are an indicator of future default, I can use monotonicity constraints to insist that my XGBoost classifier generate higher probabilities of default for higher numbers of late payments. Though I might not see gains in *in silico* test data performance with constraints, constraints do mitigate real-world stability, overfitting and sociological bias risks, and they likely increase *in vivo* performance!

Partial Dependence and Individual Conditional Explanation

Partial dependence is an established and highly intuitive post-hoc explanation method that describes the estimated average behavior of a model across the values of some input feature. Unfortunately, it's fallible. It fails to represent model behavior accurately in the presence of correlation or interactions between input features, and it can even be [maliciously altered](#). But, because understanding the average behavior of a feature in a model is so important, many techniques have been developed to address the failings of partial dependence. In particular, link: [accumulated local effect](#) (ALE) is the most direct replacement of partial dependence, and designed specifically to address the shortcomings of partial dependence. You can try ALE with packages like [ALEPlot](#) or [ALEPython](#).

In the examples below, we'll make heavy use of another partial dependence derivative to get a solid understanding of feature behavior in our models. First introduced in [Peeking Inside the Black Box: Visualizing Statistical Learning with Plots of Individual](#)

Conditional Expectation, ICE pairs plots of local behavior of the model with respect to single individuals with partial dependence. This enables us to compare estimated average behavior with descriptions of local behavior, and when partial dependence and ICE curves diverge, to decide for ourselves if partial dependence looks trustworthy or if it's looks to be affected by correlations or interactions in input variables. Of course, ICE is not without its own problems. The most common problem with ICE is the consideration of unrealistic data values, and when interpreting ICE, it's important to put the most mental weight on values of the input feature that are most similar to those in the original row of data being considered.

Let's work through all of this in an example. In left panel of Figure [Figure 6-1](#), you can see the partial dependence (red) and ICE for a penalized logistic regression model and for the input feature PAY_0, or a customer's repayment status on their most recent bill, with higher values of PAY_0 indicating greater lateness in payment. ICE curves are generated for individuals who sit at deciles of predicted probability, a good way to pick which ICE curves to plot if you're not sure where to begin.

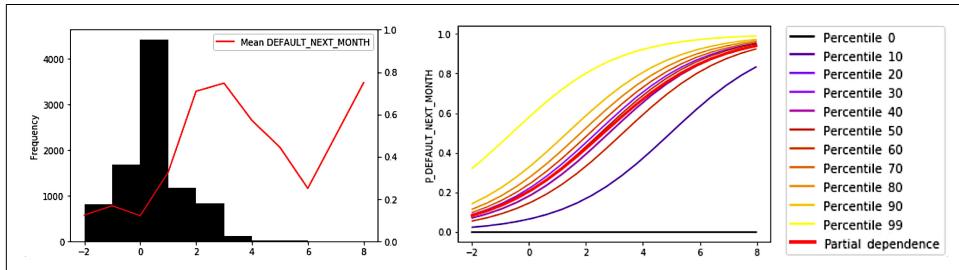


Figure 6-1. An example partial dependence plot that incorporates ICE, histograms, and conditional means to increase trustworthiness and effectiveness.

Note the smooth increase from lower probabilities of default when a customer is not late on their most recent payment, to high probabilities of default when a customer is late. This makes sense in context! It aligns with reasonable expectations and domain knowledge. Notice also that ICE and partial dependence do not diverge — they are highly aligned. This will always be the case in linear models, but it also shows us that partial dependence is likely trustworthy for this model and dataset.

In the right panel of Figure [Figure 6-1](#), we used partial dependence and ICE to understand how our model treats PAY_0. We compared partial dependence with ICE to ensure partial dependence is trustworthy. So, what's going on in the left panel of this figure?

That's were we try to decide if the model is representing our training data well and the left panel can also clue us into problems with data sparsity and prediction stability. In the left panel, the first thing you might notice is a histogram. We use that histogram to look for stability problems in model predictions. ML models typically only

learn from data, so if there is not much data, as is the case with $\text{PAY_0} > 1$ in our training data, the ML model can't learn much and their predictions in those data domains will be unstable, if not nonsensical. Some other packages use error bars for the same purpose in partial dependence or shape function plots. That's also fine. Both visualization techniques are trying to draw your eye to a region of data where your ML model is unstable and probably making silly decisions.

In the left panel, there's also a bright red line overlaid on the histogram. That red line is the conditional mean of the target for the corresponding histogram bin. If the model learned from the data correctly, the partial dependence and ICE in the right panel should roughly mirror the conditional mean line in the left panel. Sparsity is also an important caveat to keep in mind when judging whether model behavior aligns with the conditional mean of the target. In [Figure 6-1](#), we see a precipitous drop in the conditional mean at $\text{PAY_0} = 6$, or six months late in for the most recent bill. But, there's no data to support this drop. The histogram bin is basically empty and the drop is probably just irrelevant noise. Luckily our well-behaved logistic regression model has no choice but to ignore this noise and keep pushing probability of default monotonically higher as PAY_0 increases. With more complex models for the same data, we will need to apply monotonicity constraints to ensure the model follows causal relationships instead of memorizing irrelevant noise with no data support.

To quote the important explainability researcher Przemysław Biecek, “Don’t explain without context!” That means we need to think about the correlations, interactions, and the security of the datasets we’re using to generate partial dependence and ICE — typically validation, test, or other interesting holdout samples. If those datasets don’t align with the correlations and interactions in training data, or the sample could have been intentionally altered, we’ll get different results than we saw in training. This can raise a number of questions. Was the training partial dependence correct? Does my model actually behave differently in new data, or are the new correlations and interactions in this sample causing partial dependence to be less trustworthy? These are all reasons that we pair partial dependence with ICE. As a local explanation technique, ICE is less susceptible to global changes in correlations and interactions. If something looks off with partial dependence, first check if partial dependence follows the local behavior of ICE curves or if it diverges from the ICE curves. If it diverges, it’s likely safer to take your explanatory information from ICE, and if possible, to investigate what distribution, correlation, interaction, or security problems are altering the partial dependence.

Shapley Values

Because Shapley values can use *background* datasets, or specific datasets to draw from when removing features to create explanations, we have to consider [context](#) in both the dataset to be explained and the background dataset. Because of the definition of

partial dependence and ICE, we usually use very simple background datasets for these techniques. You may not even think of them as background datasets at all. We are essentially just replacing the value for an entire feature (partial dependence) or a row (ICE) with some known value of that feature in order to generate a curve. When it comes to Shapley values, we have a choice for (1) which observations we explain, anything from a single row to an entirely new sample of data, and (2) which background dataset to use when generating Shapley values, anything from not using a background set, to missing values, to random data, to highly massaged background data designed to address context or causality issues.



Recall from Chapter 2 that Shapley values are a post-hoc explanation technique, borrowed from economics and game theory, that decompose model predictions into contributions from each input feature.

In addition to considering the correlations, interactions, and security of the dataset we are explaining, we also have to ask whether our choice of background is appropriate and whether explanations make sense in the context in which they will be judged. We'll go into detail later in the chapter for how you can choose an appropriate background dataset for your Shapley value explanations, depending on the question that your explanations are trying to answer. In practice, this complex analysis often boils down to computing explanations on a few different datasets and making sure results are salient and stable. If you are computing Shapley-based explanations, it also means documenting the background dataset used and the reasons for choosing this dataset.

Model Documentation

Model documentation is the physical manifestation of accountability in large organizations — it's really about adulting at work. When we have to write a document about a model we built, knowing that our name will be ascribed to the same document, hopefully this encourages more deliberate design and implementation choices. And if we don't make sensible choices, document bad choices, or the documentation is clearly missing or dishonest, then there can be consequences for poor model building. Model documentation is also important for maintenance and incident response. When you've moved on to your next big data science job, and your older models start getting stale and causing problems, documentation enables a new set of practitioners to understand how the model was intended to work, maintain it in future iterations, and fix it when needed.

There are now several standards for model documentation, including:

- **Model Cards**, with Google-provided [example model cards](#).

- Model risk management in-depth documentation. See the [2021 model risk management guidance](#) from the United States (US) Office of the Comptroller of the Currency (OCC).
- The European Union (EU) AI Act [documentation template](#), see Document2, Appendix IV.

Notice all of these templates come from either a leading commercial user and maker of ML, or a very serious government body. If you've avoided model documentation until now, expect that to change in the future, especially for important applications of ML. All of these templates also benefit greatly from interpretable models and post-hoc explanation, because increased transparency is yet another goal and benefit of model documentation. Transparency into ML models allows us to understand, and then to justify, design and implementation trade-offs. If what you see in an interpretable model or post-hoc explanation result appears reasonable and you can write a few commonsense sentences to justify the observed outcomes, that's what we're going for in this chapter! If instead, you're using a black-box and don't understand how design and implementation tradeoffs affect model behavior, your documented justifications will likely be much weaker, and open you and your model up to potentially unpleasant external scrutiny.

The GAM Family of Interpretable Models

In this [example](#), we'll form a baseline with a linear additive penalized regression model, then compare that baseline to a GAM that allows for complex non-linearity, but in an independent, additive, and highly explainable fashion. We'll then compare the GLM and GAM to an EBM with a small number of two-way interactions. Because all of our models are constructed with additive independent functional forms, and because we will use only a small number of meaningful interactions, all our models will be very explainable. Additivity will enable us to make clear and justifiable choices about introducing nonlinearity and interactions.

Elastic Net Penalized GLM w/ Alpha and Lambda Search

As the name suggests, a generalized linear models (GLMs) extend the idea of an ordinary linear regression and generalizes for error distributions belonging to exponential families, in addition to the Gaussian distribution of error used in standard linear regression. Another vital component of a GLM is the link function that connects the expected value of the response to the linear components. Since this link function can be any monotonic differentiable function, GLMs can handle a wide variety of distributions of training data outcome values: linear, binomial — as in our current example, Poisson, and several others. Penalizing a GLM refers to using sophisticated constraints and iterative optimization methods to handle correlations, feature selec-

tion, and outliers. Putting all these together results in a robust modeling technique with good predictive power and very high interpretability.

Elastic net is a popular regularization technique that combines the advantages of both L1 (LASSO) and L2 (ridge) regression into one. While the L1 regularization enables feature selection, thereby inducing sparsity and higher interpretability in the trained model, L2 regularization effectively handles correlation between the predictors. The iteratively reweighted least squares (IRLS) method is often paired with elastic net to handle outliers as well.

Training a penalized GLM will serve two useful benchmarking purposes:

- Since our GLM does not include any non-linearity or feature interactions, it can serve as the perfect benchmark to test certain hypotheses. i.e., whether non-linearities and interactions actually result in a better model or not, which we'll cover in the upcoming sections.
- The GLM also acts as a starting point for initial feature selection based on the features selected by L1 regularization.

We'll start our first example for Chapter 6 by training an elastic net penalized logistic regression using H2O's GLM algorithm, which works in a distributed fashion and scales well for large datasets. In H2O GLM, the regularization parameters are denoted by alpha and lambda. While alpha specifies the regularization distribution between L1 and L2 penalties, lambda indicates the regularization strength. The recommended way to find optimal regularization settings in H2O GLM is via a grid search. H2O offers two types of grid searches — Cartesian and random search. Cartesian is an exhaustive search that tries all the combinations of model hyperparameters specified in a grid of possible values by the user. On the other hand, random grid search samples sets of model parameters randomly from the given set of possible values based on some stopping criterion. By default, H2O will use Cartesian search, and we'll use that for our use case because it won't take too long to search over a small number of alpha values.

In the code block below, we start by defining a grid of model hyperparameters for alpha values. It is important to note here that to preserve the stabilizing functionality of L2 penalties and the feature selection functionality of L1 penalties, alpha should never be 0 or 1. This is because when alpha is 0, it denotes only the L2 penalty, while a value of 1 for alpha signifies only L1. H2O's GLM implementation comes with a handy `lambda_search` option. When set to `True`, as below, this option searches over various lambda values starting from `lambda_max` with no features in the model to `lambda_min` with very small values of a lambda with many features. Both alpha and lambda are selected by validation-based early stopping. This means the GLM will automatically stop building the model when there is no significant improvement on the validation set, and automatically limit overfitting.

```

def glm_grid(x, y, training_frame, validation_frame, seed_, weight=None):

    """
    :param x: List of inputs.
    :param y: Name of target variable.
    :param training_frame: Training H2OFrame.
    :param validation_frame: Validation H2OFrame.
    :param seed_: Random seed for better reproducibility.
    :return: Best H2OGeneralizedLinearEstimator.

    """

    # settings GLM grid parameters
    alpha_opts = [0.01, 0.25, 0.5, 0.99] # always keep some alpha!
    hyper_parameters = {'alpha': alpha_opts}

    # initialize grid search
    glm_grid = H2OGridSearch(
        H2OGeneralizedLinearEstimator(family="binomial",
                                       lambda_search=True,
                                       seed=seed_),
        hyper_params=hyper_parameters)

    # training w/ grid search
    glm_grid.train(y=y,
                   x=x,
                   training_frame=training_frame,
                   validation_frame=validation_frame,
                   weights_column=weight,
                   seed=seed_)

    # select best model from grid search
    best_model = glm_grid.get_grid()[0]
    del glm_grid

    return best_model

```

Using the function above to run a Cartesian search over alpha, and letting H2O search over the best lambda values, our best GLM ends up with an AUC score of 0.73 on the validation dataset. The six PAY_*` repayment status features are selected among all other features to predict DELINQ_NEXT.



An AUC score of 0.73 means there is a 73% chance our model will properly rank a randomly drawn positive row with a higher output probability than that of a randomly drawn negative row.

To understand how the model treats various features, we plot partial dependence in conjunction with ICE plots for the features of interest. Additionally, a histogram of

the feature of interest, including an overlay of the mean value of the target column i.e. `DELINQ_NEXT` is displayed alongside. This should give a good idea of whether the model is behaving reasonably and if any data sparsity issues could result in meaningless predictions.

Let's revisit [Figure 6-1](#) The `PAY_0` feature has the steepest partial dependence and ICE curve, thereby suggesting it's the most important input feature. The partial dependence and ICE plots are in harmony, i.e., they do not diverge, implying that the partial dependence can be relied upon. Additionally, there is a monotonic increasing relationship of predicted probability of default and `PAY_0` lateness of payment. This means as the delay in payment increases, the probability that a customer will default also becomes larger. This is in line with our intuition of how credit card payments work.

Now let's review the histogram on the left. For customers with late payments, there are some apparent data sparsity problems. For instance, in regions where $\text{PAY}_0 > 1$ there is little or no training data. Also, the mean `DELINQ_NEXT` values do exhibit some non-linear patterns in this region. It is all but obvious that predictions made in these regions will be less trustworthy. After all, a model can only learn from the data, unless we provide it additional domain knowledge. However, the good news is that the logistic form of our penalized GLM not only prevents it from being fooled by low-confidence dips in conditional mean `DELINQ_NEXT` around $\text{PAY}_* = 6$, and not overfitting noise in these areas of sparse training data. The model treats the other `PAY_*` features similarly, but assigns them flatter logistic curves. In all cases, probability of default increases monotonically with the lateness of payment, as expected. To see the other partial and dependence and ICE plots, check out the [code resources](#) for Chapter 6.

We now have a robust and interpretable baseline model. Because its behavior makes so much sense and is so easy to interpret, it may be hard to beat. A validation AUC of 0.73 is nothing remarkable, but having an interpretable model that behaves in a manner that aligns to time-tested causal relationships that we can count on once deployed — that's priceless for risk mitigation purposes. We also have to remember that validation and test data assessment scores can be misleading in more complex ML models. We can have a high AUC in static validation or test data, just to find out that high AUC arose from overfitting to some specific phenomenon that's no longer present in our operational domain. In the next section, we'll first introduce some non-linearities via GAMs and then feature interactions in addition to the non-linearities via EBMs. We'll then assess our model for interpretability and performance quality with an eye toward real-world performance. We'll try do honest experiments and make deliberate choices about whether more complexity is justified.

Generalized Additive Models

While linear models are highly interpretable, they cannot accurately capture the non-linearities typically present in real-world datasets. This is where generalized additive models (GAMs) come into play. GAMs, originally **developed at Stanford in the late 1980s** by eminent statisticians Trevor Hastie and Rob Tibshirani, model nonlinear relationships of each input feature with individual spline shape functions and adds them all together to make a final model. GAMs can be thought of as additive combinations of spline shape functions. An important idea for GAMs is, even though we treat every feature in a very complex way, it is done in an additive and independent manner. This not only preserves the interpretability but also enables editing and debugging with relative ease.

When it comes to implementing GAMs, packages like `gam` and `mgcv` are some great options in R. As for Python, the choice is limited, as most of the packages are in the experimental phase like H2O's `GAM` implementation. Another alternative, `pyGAM`, derives its inspiration from R's `mgcv` package has been shown to offer a **good combination of accuracy, robustness, and speed** and has a familiar scikit-like API.

We'll use `pyGAM` to train a GAM model on the same credit card dataset we used in the last section. Specifically, we'll implement a `LogisticGAM` using `pyGAM`'s `Logisti cGAM` class. There are three important parameters that can be tuned to obtain the best model: number of splines, the lambda or the strength of regularization penalty, and constraints term to inject prior knowledge into the model. `pyGAM` provides an inbuilt grid search method to search over the smoothing parameters automatically.

```
from pygam import LogisticGAM
gam = LogisticGAM(max_iter=100, n_splines=30)
gam.gridsearch(train[features].values, train[target], lam=np.logspace(-3, 3,
15))
```

The code above instantiates a `LogisticGAM` model that will train for up to 100 iterations. The `n_splines` parameter specifies the number of spline terms, or the complexity of the function used to fit each input feature. More spline terms typically results in more complex spline shape functions. `lam` corresponds somewhat to `lambda` in penalized regression, and the code above searches over several values to find the best strength of regularization as defined by `lam`. One parameter we are not taking advantage of is `constraints`. `constraints` allows users to specify a list of constraints for encoding prior knowledge. The available constraints are monotonic increasing or decreasing smoothing and convex or concave smoothing. We'll work with similar constraints later in the chapter and it's very instructive to see what not using constraints means for our GAM model.

A question that we're trying to answer deliberately in this example is: are such nonlinearities truly helpful, or just overfitting noise? Many data science practitioners today assume more complexity results in better models, but we'll use GAMs to run an

experiment to decide whether introducing nonlinearity actually improves our model, from both a performance quality perspective and an interpretability perspective.

After we train our model, we calculate its validation AUC, which turns out to be 0.75 -- a notch higher as compared to our penalized GLM model. The increase in GAM AUC can likely be attributed to the introduction of non-linearity, which GLM failed to capture. However, it is important to note here that higher AUC doesn't always guarantee better models, and this example is a classic case to prove the point. We spent a bit of time analyzing how GLM treats `PAY_0`, or a customer's most recent repayment status feature in the last section, and it did a reasonably good job. Let's now look at how the GAM treats the same `PAY_0` feature.

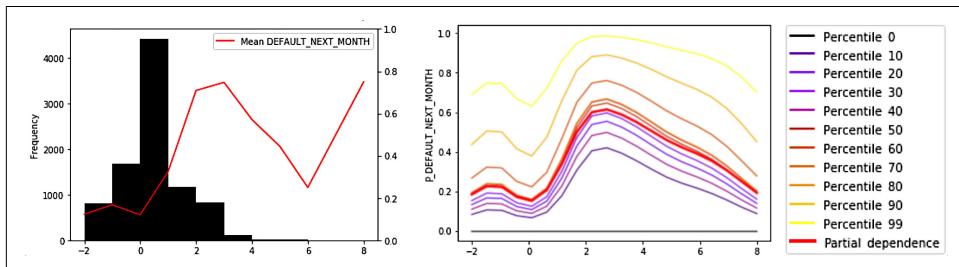


Figure 6-2. A partial dependence plot that incorporates ICE, histograms, and conditional means to increase trustworthiness and effectiveness for `PAY_0` in the example GAM.

Figure 6-2 shows that there is clearly some weirdness in the PDP and ICE plots generated via a GAM. We observe that as the lateness of payment increases, the chances that a customer will default on the payment decreases. This is obviously not correct. Most people don't magically get more likely to pay bills after months of being late. The same strange behavior is also observed for `PAY_4` and `PAY_6`, as can be seen in **Figure 6-3**. `PAY_4` probability of default looks to decrease as payment lateness increases, and `PAY_6` appears to bounce noisily around a mean prediction. Both modeled behaviors are counter-intuitive, both contradict the GLM baseline model, and both fail to model the conditional mean behavior displayed on the right hand side of **Figure 6-3**.

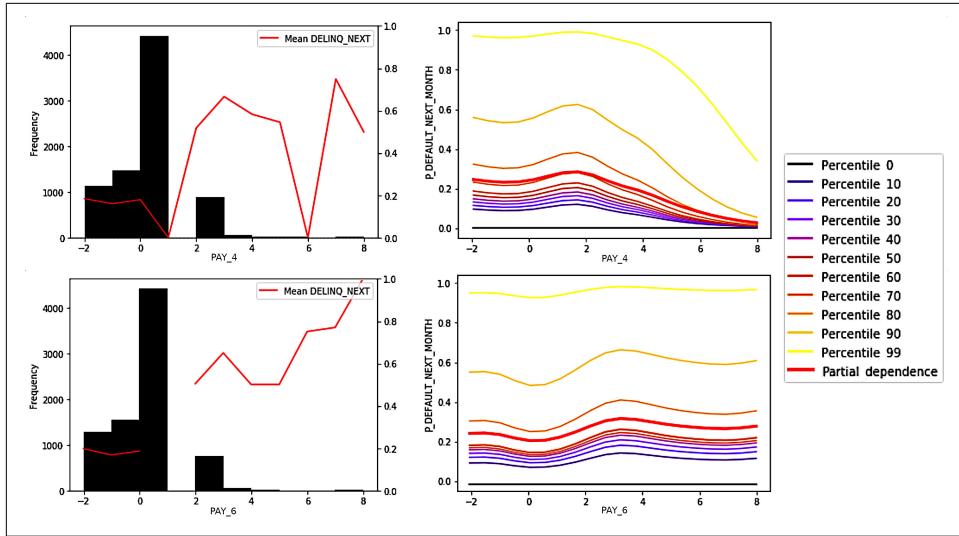


Figure 6-3. A partial dependence plot that incorporates ICE, histograms, and conditional means for PAY_4 and PAY_6.

The bottom line is that although our validation AUC is higher, this is definitely a model that we wouldn't want to deploy. As is apparent from Figure 6-2 above, the GAM has either overfit noise in the training data, been fooled by the low-confidence dips in the conditional mean DELINQ_NEXT around PAY_* = 6, or reverting to a mean prediction due to data sparsity for PAY_0 > 1.

So what's the workaround, and how do you use such a model? Well, that's precisely where the GAMs shine. The behavior displayed by the GAM, in this case, is a general problem observed for high-capacity non-linear models. However, unlike many other types of ML models, GAM not only highlight such inconsistencies but also offers ways to debug them through commonsense model editing. More plainly, we can discuss the GAM results with domain experts, and if they agree with the more plausible GAM splines for PAY_2, PAY_3, and PAY_5, we could keep those in the model and perhaps gain a boost in the model performance. As for the obviously problematic splines for PAY_0, PAY_4, and PAY_6, they can be replaced with something that makes more sense. One option is their learned behavior from the logistic regression model as shown in the expression below:

$$\begin{aligned}\hat{P} = \beta_0 + \frac{1}{1 + \exp \beta_{PAY_0, GLM} PAY_0} + \beta_{GAM, PAY_2} g(PAY_2) + \beta_{GAM, PAY_3} g(PAY_3) \\ + \frac{1}{1 + \exp \beta_{PAY_4, GLM} PAY_4} + \beta_{GAM, PAY_5} g(PAY_5) + \frac{1}{1 + \exp \beta_{PAY_6, GLM} PAY_6} + \dots\end{aligned}$$

where β_0 is an intercept term and each g represents a GAM spline function. Also note that edited behavior could also be applied to certain regions of problematic behavior, and need not fully replace a learned spline.

Edit-ability is a great feature for a predictive model, but you also need to be careful with it. If we were to edit a custom model as suggested above, it really needs to be stress-tested more than usual. Let's not forget, the coefficients weren't learned together and may not account for one another well. There could also be boundary problems — the edited model could easily result in predictions above 1 and below 0. Another, potentially more palatable debugging strategy, is to use the constraint functionality provided by PyGAM. A positive monotonic constraint would likely fix the problems in the PAY_0, PAY_4, and PAY_6 splines.

Whether we choose to edit the example GAM or retrain it with constraints, we'll likely see lower validation and test data performance quality. But, when we're most concerned with dependable real-world performance, we sometimes have to give up worshiping at the alter of holdout dataset performance. While model editing may sound strange, the model above makes sense. What seems more strange to us is deploying models whose behavior is only justified by a few rows of high-noise training data in obvious contradiction of decades of causal norms. We'd argue the model above is much less likely to result in a catastrophic failure than the nonsense splines learned by the unconstrained GAM.

This is just one of many scenarios where traditional model assessment can be misleading for choosing the best real-world model. As shown in the GAM example, we can't assume that non-linearities make better models. Moreover, GAMs allow us to test the implicit hypothesis that nonlinearity is better. With GAMs, we can create a model, interpret and analyze the results, and then edit or debug any detected issues. GAMs help us uncover what our model has learned, keep the correct results, and edit and rectify the wrong ones so that we do not deploy risky models.

GA2M and Explainable Boosting Machines

When a small group of interacting pairs of features is added to a standard GAM, the resulting model is called a GA2M or generalized additive model plus two-way interactions. Adding these pairwise interactions to traditional GAMs has been shown to substantially increase model performance while retaining interpretability, as discussed in Chapter 2. Additionally, just like GAMs, GA2Ms are also easily editable.

EBM is a fast implementation of the GA2M algorithm by Microsoft Research. The shape functions in an EBM are fit by boosted tree methods, thereby making it very robust while having accuracy comparable to the tree-based models like random forest and XGBoost. EBM comes packaged within a broader ML toolkit called **interpret**, an open-source package for training interpretable models and explaining black-box systems.

We continue with our credit card example and train an EBM to predict which customer has a high chance of defaulting on their next payment. The EBM model achieves a validation AUC of 0.78, which is the highest compared to traditional GAM and GLM. The bump in the accuracy is likely due to the introduction of nonlinearity and interactions. Interpreting EBMs and GA2Ms is also easy. Like traditional GAMs, we can plot the shape functions of individual features and their accompanying histograms that describe the model behavior and data distribution for that feature, respectively. The interaction terms can be rendered as a contour plot on a two-dimensional plane — still easy to understand! Let's dig in a bit more by looking at how EBM treats `LIMIT_BAL`, `PAY_0`, and `PAY_2` features as shown in [Figure 6-4](#) below.



Figure 6-4. Three important input features and an interaction feature with accompanying histograms for an Explainable Boosting Machine (EBM).

In [Figure 6-4](#), we can see three standard shape function plots for `LIMIT_BAL`, `PAY_0`, and `PAY_2`, but we can also see a contour plot for the `PAY_0` × `PAY_2` interaction. Each of these plots, even the slightly more complex contour plot, allows humans to check the behavior of the model, and when needed, to edit it. The behavior of `LIMIT_BAL` looks reasonable, as increasing credit limits are expected to be correlated with decreasing probability of default. The EBM treats `PAY_0` more logically than our GAM. Under the EBM, `PAY_0` probabilities of default increase for $\text{PAY}_0 > 1$ and stay high. `PAY_2` does appear somewhat noisy. And the interaction term exhibits the same unrealistic behavior that was observed under the GAM for some individual `PAY_*` features, where increased lateness results in lower probability of default. This is likely due to data sparsity. At least this strange behavior is plainly obvious, and `PAY_2` and `PAY_0` × `PAY_2` may be good candidates for model editing.

There are two other very important aspects of [Figure 6-4](#) that are not necessarily characteristics of EBMs, but that also require a second look — the shaded regions around the shape functions and the histograms beneath the shape functions. Both of these features help users decide the level of trustworthiness for the model. If the histograms indicate that little training data was available in a certain region, or the shaded error bars show that the function has high variance in a certain training data domain, that part of the function is probably less trustworthy and model editing can be considered.

One additional slightly tricky aspect of working with EBM is accessing information for your own plotting needs. While EBM provides amazing interactive plotting capabilities out-of-the-box, we often like to create our own plots or data structures, especially to compare with other models. We found it necessary to interact with EBM's `_internal_obj` JSON structure to do so on several occasions. Take, for instance, accessing feature importance values as below.

```
ebm_global = ebm.explain_global(name='EBM')
feature_names = ebm_global._internal_obj['overall']['names']
feature_importances = ebm_global._internal_obj['overall']['scores']
ebm_variable_importance = pd.DataFrame(zip(feature_names, feature_importances),
                                         columns=['feature_names', 'feature_impor-
tance'])
```

To extract feature importance to manipulate ourselves, instead of relying on the EBM's default plotting, we had calculate global explanations using `explain_global()`, then extract feature names and importance scores from JSON within the returned object. We then used this information to create a Pandas Data Frame, and from there, most standard operations like plotting, selecting, or manipulating, are easy.

With that, we'll close out our first set of examples. In this section, we introduced a benchmark GLM, then deliberately introduced nonlinearities via GAM and interactions via GA2M and EBM that made our models more complex. However, due to the additive nature of GLMs, GAMs, and EBMs, not only did we retain interpretability and gain performance quality, we also created a set of editable models that we can compare to one another, and even combine, to build the best possible model for real-world deployment. The next section will continue these themes and dive into constraints and XAI with XGBoost.

XGBoost with Constraints and Explainable Artificial Intelligence

In this [example](#), we'll train and compare two XGBoost classifier models - one with monotonic constraints, and one without. We'll see that the constrained model is more robust than the unconstrained and no less accurate. Then, we'll examine three power-

ful *post-hoc* explanation methods - decision tree surrogates, partial dependence and ICE, and SHAP values. We'll conclude with a technical discussion of shap value calculations and background datasets, and we'll provide guidance so you can choose the appropriate specifications for your application.

Constrained and Unconstrained XGBoost

XGBoost is an incredibly popular model architecture for prediction tasks on large, structured datasets. So what is an XGBoost model? The models produced by XGBoost are ensembles of *weak learners*. That is, XGBoost produces a bunch of small models in sequence, and then to make a final prediction it sums up the predictions of these small models. In this section, we'll use XGBoost to train an ensemble of shallow decision trees. We'll be working with a binary classification problem, but XGBoost can be used to model other problem types, such as regression, multiclass classification, survival time, and more.

XGBoost is so popular, in part, because it tends to produce robust models that generalize well on unseen data. That doesn't mean we, as model developers, can fall asleep at the wheel! We have to use reasonable hyperparameters and techniques such as early stopping to ensure that this strength of the model architecture is realized. Another thing that XGBoost allows us to do is impose monotonic constraints on our models. These constraints lock down the direction of the relationship between certain features and the model output. They allow us to say - *If feature X_1 increases, then the model output cannot decrease*. In short, these constraints allows us to apply our own domain knowledge, and even causal knowledge in some cases, to make more robust models. Let's take a look at some code to train an XGBoost model below.

```
params = {  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
    'eta': 0.05,  
    'subsample': 0.75,  
    'colsample_bytree': 0.8,  
    'max_depth': 5,  
    'base_score': base_score,  
    'seed': seed  
}  
  
watchlist = [(dtrain, 'train'), (dvalid, 'eval')]  
  
model_unconstrained = xgb.train(params,  
                                 dtrain,  
                                 num_boost_round=200,  
                                 evals=watchlist,  
                                 early_stopping_rounds=10,  
                                 verbose_eval=True)
```

First, let's look at the values in the `params` dictionary. The parameter `eta` is the *learning rate* of our model. The smaller value of `eta`, the less weight is given to individual decision trees that occur later in the sequence. If we used `eta = 1.0`, our model's final prediction would be an unweighted sum of individual decision tree outputs, and would almost certainly overfit to the training data! Make sure you set a reasonable learning rate (say, between 0.001 and 0.3) when training XGBoost or other gradient boosting models.

The parameters `subsample` and `colsample_bytree` also protect against overfitting. Both ensure that each individual decision tree does not see the entire training dataset. In this case, each tree sees a random 75% of rows of the training data (`subsample = 0.75`), and a random 80% of columns of the training data (`colsample_bytree = 0.8`). Then, we have some parameters that dictate the size of the final model. `max_depth` is the depth of the trees in the model. Deeper trees include more feature interactions and are more complex than shallow trees. You usually want to keep your trees shallow when training XGBoost and other GBM models - after all, the strength of these models comes from them being an ensemble of weak learners, i.e., the wisdom of the crowd! Of course, grid search and other structured methods for selecting hyperparameters are the better practice for choosing these values, but that's not the focus of this chapter.

Finally, in the above code snippet we're training a model using validation-based `early stopping`. We do this by feeding a dataset (or in this case, two datasets) into the `evals` parameter, and by specifying `early_stopping_rounds`. So what is going on here? Remember that XGBoost trains decision trees in sequence, one after another. Each *round* of this training sequence, the collection of decision trees trained so far is evaluated on the datasets in the `evals` watchlist. If the evaluation metric (in this case, AUC) does not increase for `early_stopping_rounds` rounds, then training stops. If we didn't specify early stopping, then training would proceed until `num_boost_round` trees had been built - but this stopping point is often arbitrary! You should always use early stopping when training your GBM models.



If you pass multiple datasets into `evals`, only the *last* dataset in the list will be used to determine if the early stopping criterion has been met.



If you train a model with early stopping, the final model has too many trees! Whenever you make a prediction with the model, you probably should specify how many trees to use using the `iteration_range` parameter - see [the documentation](#) for more info.

Now that we've trained our *unconstrained* model - as we'll see below, XGBoost was free to assign probabilities to individual observations based on, sometimes spurious, patterns in the training data. We can likely do much better using the knowledge in our own brains in addition to what can be learned from training data.

We know, for example, that if someone is more and more overdue on their credit card payments then they almost certainly have a higher likelihood of being late on their next payment. That means for all PAY_* features in our dataset, we'd like the model output to increase when the feature value increases, and vice versa. XGBoost monotonic constraints allow us to do exactly that. For each feature in the dataset, we can specify if we'd like that feature to have a positive or negative monotonic relationship with the model output.

Our dataset only contains 19 features, and we can reason through the underlying causal relationship with default risk for each one. What if your dataset contains hundreds of features? You'd like to train a robust, constrained model, but maybe you're unsure about a monotonic causal link between certain features and the target. An alternative (or supplemental) method to deriving monotonic constraints uses Spearman correlation. In the below code, we implement a function that examines the pairwise Spearman correlation coefficient between each feature and the target. If the Spearman coefficient is greater in magnitude than a user-specified threshold, that feature is assumed to have a monotonic relationship with the target. The function returns a tuple containing values -1, 0, and 1 - exactly the form of input expected by XGBoost when specifying monotonic constraints.

```
def get_monotone_constraints(data, target, corr_threshold):
    corr = pd.Series(data.corr(method='spearman')[target]).drop(target)
    monotone_constraints = tuple(np.where(corr < -corr_threshold, -1,
                                           np.where(corr > corr_threshold, 1,
                                           0)))
    return monotone_constraints
```



We use Spearman correlation coefficients rather than the default Pearson correlation because GBMs are nonlinear models, even when constrained. XGBoost monotonic constraints impose *monotonicity* rather than *linearity* - the Spearman correlation measures exactly the strength of a monotonic relationship, whereas the Pearson correlation measures the strength of a linear relationship.

In [Figure 6-5](#) below, we've plotted the Spearman correlation coefficient of each feature against the target variable. The vertical lines indicate the threshold value of 0.1. We can see that this data-driven approach suggests imposing monotonic constraints for the PAY_* features. We're using the threshold of 0.1 as an informal marker of practical significance. We may not want to apply any constraints to those input features whose Spearman correlation is less than 0.1 in magnitude.

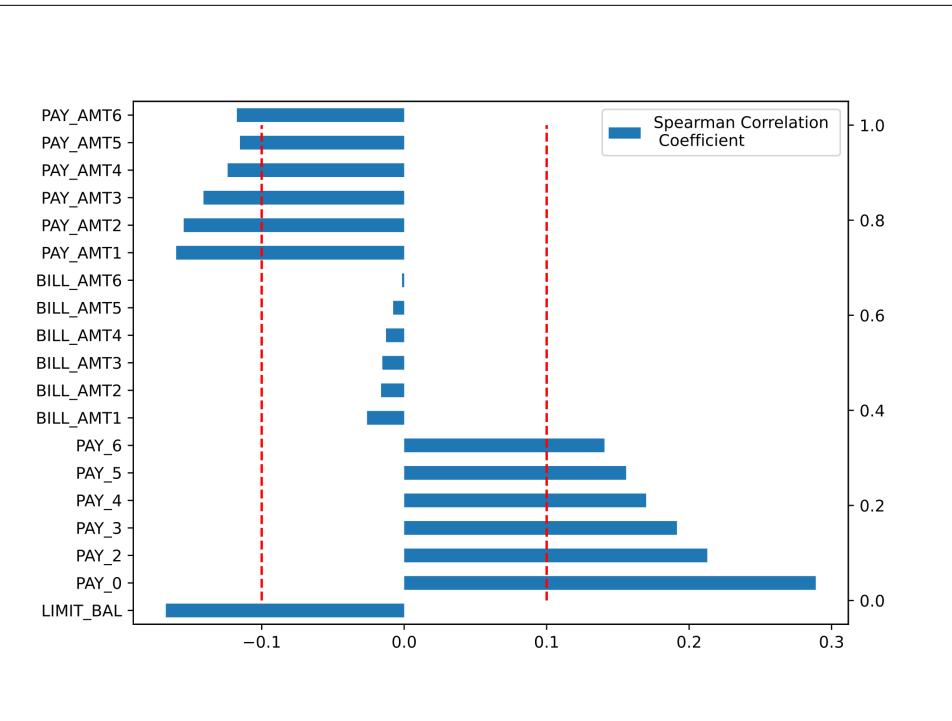


Figure 6-5. The Spearman correlation coefficients between target `DEL_INQ_NEXT` and each feature, with vertical bars indicating a cutoff value of 0.1.

Next, we train the constrained model with the constraints suggested by the analysis above. Let's make a few observations about the resulting constrained and unconstrained models.

By looking at the output of `xgb.train()` with `verbose_eval=True` in the code [example](#), we see that the unconstrained model has a higher AUC on the training set (0.818 vs 0.807), but shows equal performance to the constrained model on the validation set (0.784 vs 0.783). This suggests that the constrained model is less overfit than the unconstrained model - without the exact same set of hyperparameters, the constrained model picks up on a higher proportion of the true signal in the data. As our analyses will show, there are additional reasons that we expect better performance (and better stability) from the constrained model *in vivo*.

Finally, let's look at the feature importance for the two models in [Figure 6-6](#). You can compute the feature importance values for XGBoost models in many ways. Here, we'll look at the average coverage of the splits in the ensemble. The coverage of a split is just the number of training samples that flow through the split. While this is a useful proxy for feature importance, it does not have the same theoretical guarantees as, for example, Shapley-value-based techniques.

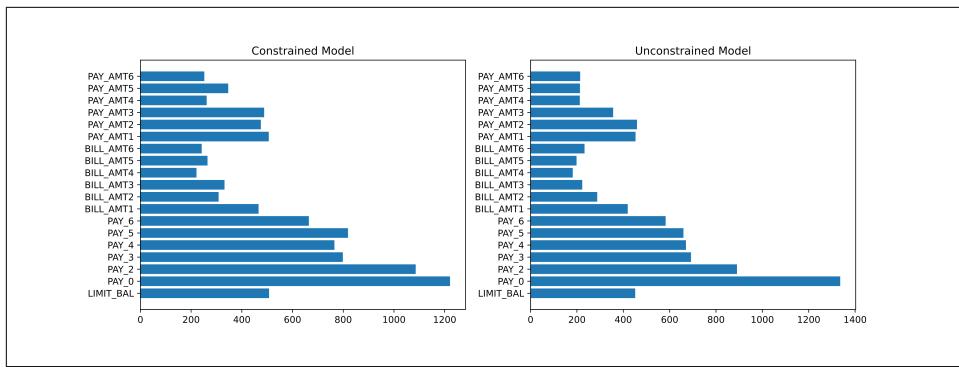


Figure 6-6. The feature importance values of the constrained and unconstrained models, as measured by average coverage.

You can see that the constrained model spreads the feature importance more evenly among the entire input feature set. The unconstrained model gives a disproportionate share of the feature importance to the PAY_0 feature. This is even more evidence that the constrained model will be more robust when deployed. If a model focuses all its decision-making capacity on one feature, then it's going to fail when the distribution of that feature drifts in new data. Being too dependent on a single feature is also a security risk. It's easier for bad actors to understand how the model works and take advantage of it!



When you see feature importance values concentrated on one or a few features, your model is more likely to be unstable and insecure post-deployment. Your model might be overly sensitive to drift in the data distribution along a single dimension, and a malicious actor only needs to manipulate the value of one feature to alter model outcomes.

Explaining Model Behavior with Partial Dependence and ICE

Let's continue our comparison of the constrained and unconstrained XGBoost models by looking at a side-by-side partial dependence and ICE plot for PAY_0. In the previous sections, we've already discussed how the conditional mean of the target variable shows a spurious dip around PAY_0 = 6, where our training data is sparse. Let's see how our two XGBoost models handle this data deficiency.

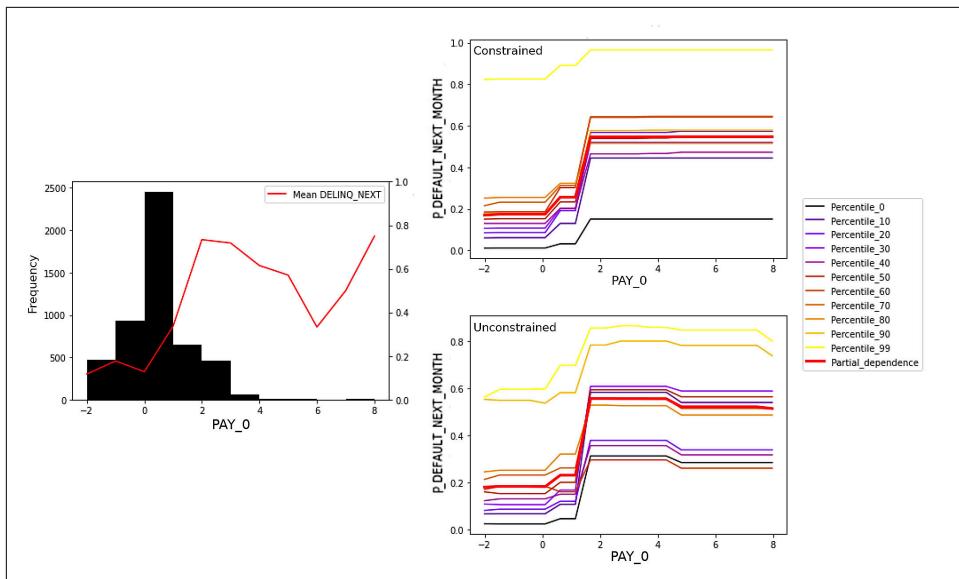


Figure 6-7. Partial dependence and ICE plots of the PAY_0 feature for the constrained (top) and unconstrained (bottom) models.

In [Figure 6-7](#) above, we can see how the unconstrained model overfits to the spurious relationship between PAY_0 and DELINQ_NEXT. On the other hand, the constrained model is forced to obey the commonsense relationship that more delayed payments lead to a higher risk of delinquency. This is reflected in the monotonically increasing partial dependence plot for PAY_0 under our constrained model!

You can also see that for both models, there are large changes in the model output across the range of PAY_0 values. That is, both the partial dependence and ICE plots show lots of vertical movement as we sweep PAY_0 from -2 to 8. The model outputs are highly sensitive to the values of this feature - which is exactly why we saw such high feature importance values for PAY_0 in [Figure 6-6](#). If you don't observe these kinds of value changes for a feature your model says is very important, that's a sign more debugging may be required.

Partial dependence and ICE plots can also reveal where there are feature interactions in our model. Take a look at the partial dependence and ICE plot for LIMIT_BAL under our unconstrained model in [Figure 6-8](#).

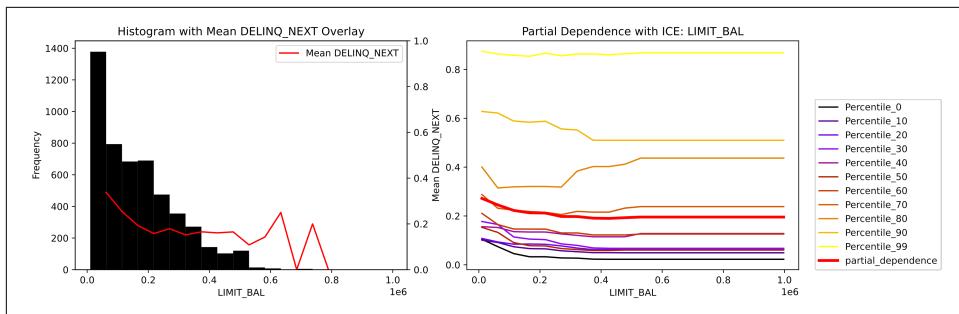


Figure 6-8. Partial dependence and ICE plots of the LIMIT_BAL feature for the unconstrained model.

As discussed in the concept refresher, when partial dependence and ICE curves diverge, as they do here, it is suggestive of correlations or interactions in our data and model. Moreover, we can look back to the link:[EBM training](#) and see that two important interactions identified by the EBM model are LIMIT_BAL`x`BILL_AMT2 and LIMIT_BAL`x`BILL_AMT1. It's likely that our unconstrained XGBoost model picked up on these interactions as well. In contrast to the EBM, our XGBoost models are riddled with various high-degree feature interactions. But partial dependence and ICE, combined with the the EBM's ability to learn two-way interactions, can help us understand some of the interactions in our XGBoost models too. Another great tool for understanding complex feature interactions in ML models is the surrogate decision tree, which we'll discuss next.

Decision Tree Surrogate Models as an Explanation Technique

The analysis we've conducted so far has shown that our constrained XGBoost model is no less accurate than the unconstrained version. The partial dependence and ICE plots we've looked at show that by tethering the constrained model to reasonable real-world relationships, we've successfully prevented the model from picking up on spurious relationships in the training data. Since our constrained model appears logically superior to the alternative, the next sections will focus exclusively on this model.

First, we're going to continue our exploration of the model's behavior through a post-hoc explanation technique - decision tree surrogate models. A *surrogate model* is just a simple model meant to mimic the behavior of a more complex model. In our case, we're trying to mimic our constrained XGBoost model with roughly 100 trees using a single, shallow decision tree. A decision tree is a data-derived flowchart, so we can look at the decision tree surrogate flow chart and explain how it is operating in simple language. This is what makes decision tree surrogates a powerful explanation technique. We use the `DecisionTreeRegressor` implementation from `sklearn` to train our surrogate model:

```

surrogate_model_params = {'max_depth': 4,
                           'random_state': seed}
surrogate_model = DecisionTreeRegressor(**surrogate_model_params)
    .fit(train[features], model_constrained.predict(dtrain))

```

Notice we're training a regression model targeted at the output of the model we're trying to explain. That is, the surrogate is totally focused on mimicking the behavior of the larger model, not just making a simpler classification model. We've also chosen to train a decision tree of depth four. Any deeper, and we might have a hard time explaining what is happening in the surrogate model itself!

Before we examine our surrogate model, we need to ask whether we can trust it. Decision tree surrogates are a powerful technique, but they don't come with many mathematical guarantees. One simple way of assessing the quality of our surrogate is to compute accuracy metrics on cross-validation folds. Why cross-validation and not just a validation dataset? One of the pitfalls of single decision tree models is their sensitivity to changes in the training dataset, so by computing accuracy on multiple holdout folds, we're checking whether our surrogate model is accurate and stable enough to trust.



The technique we demonstrate here is a simple way to train and evaluate a decision tree surrogate. The research community has created other methods to train surrogate decision trees, such those discussed in *Interpreting Blackbox Models via Model Extraction* and *Extracting Tree-Structured Representations of Trained Networks*. There has also been some work on what mathematical guarantees we can make about surrogate models, such as in the first reference, as well as in *The Price of Interpretability*.

```

from sklearn.model_selection import KFold
from sklearn.metrics import r2_score

cross_validator = KFold(n_splits=5)
cv_error = []
for train_index, test_index in cross_validator.split(train):
    train_k = train.iloc[train_index]
    test_k = train.iloc[test_index]

    dtrain_k = xgb.DMatrix(train_k[features],
                           label=train_k[target])
    dtest_k = xgb.DMatrix(test_k[features],
                           label=test_k[target])

    surrogate_model = DecisionTreeRegressor(**surrogate_model_params)
    surrogate_model = surrogate_model.fit(train_k[features],
                                         model_constrained.predict(dtrain_k))
    r2 = r2_score(y_true=model_constrained.predict(dtest_k),
                  y_pred=surrogate_model.predict(test_k[features]))
    cv_error.append(r2)

```

```
cv_error += [r2]

for i, r2 in enumerate(cv_error):
    print(f"R2 value for fold {i}: {np.round(r2, 3)}")
print(f"\nStandard deviation of errors: {np.round(np.std(cv_error), 5)}")

R2 value for fold 0: 0.900
R2 value for fold 1: 0.907
R2 value for fold 2: 0.918
R2 value for fold 3: 0.894
R2 value for fold 4: 0.901

Standard deviation of errors: 0.00799
```

These results look great! You can see that the surrogate model has a high accuracy across every cross-validation fold, with very little variation. Confident that our decision tree is a reasonable surrogate, let's plot the surrogate model:

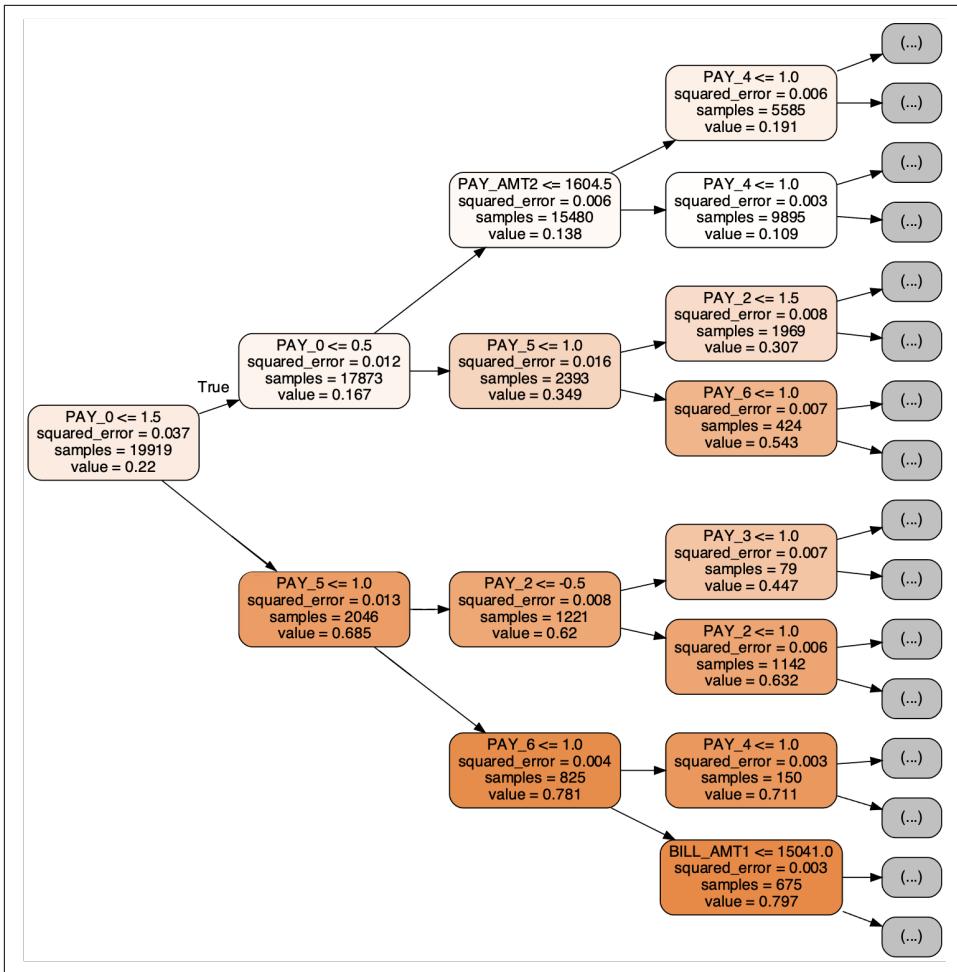


Figure 6-9. The decision tree surrogate for our constrained XGBoost model.

Notice in Figure 6-9 above that the the surrogate model splits on the PAY_0 feature first. That is, in order to optimally mimic the behavior of the constrained XGBoost model, the first thing the surrogate model does is to separate observations into two groups - those with PAY_0 = -1, 0, or 1, and those with higher PAY_0 values. We can crudely approximate feature importance by looking at the depth of each features' splits in the surrogate model, so this result is consistent with our feature importance analysis above. A good sign!

Since our surrogate model is so simple, we can make lots of plain-English observations about it. For example, I can trace the paths of the highest- and lowest-risk observations and explain how our surrogate is treating them. The lowest-risk observations traverse the following decision path: *September, 2005 repayment status on*

time or one month late ($PAY_0 \leq 1.5$) AND August, 2005 repayment status on-time or one month late ($PAY_2 \leq 1.5$) AND amount paid in August, 2005 greater than \$1,500 AND paid on-time in June, 2005 ($PAY_4 \leq 0.5$). We can do the same thing for the highest-risk path: September, 2005 repayment status more than one month late ($PAY_0 > 1.5$) AND April, 2005 repayment status more than one month late ($PAY_6 > 1$) AND amount paid in July, 2005 less than \$14,500 AND May, 2005 repayment status more than one month late ($PAY_5 > -0.5$). Both of these explanations take into account recent payments and payments in the past. Our models are looking for payment behavior patterns over time, which does makes sense.

The last thing we'll notice is that each time one feature follows from another in a decision tree path, those features are likely interacting. We can easily identify the main feature interactions that our XGBoost model has learned by examining the surrogate model. Interestingly, we can also look back and see that the EBM also picked up on many of the same interactions, such as $PAY_0 \times PAY_2$ and $PAY_0 \times PAY_4$! With all these tools — EBMs, partial dependence and ICE, and surrogate decision trees — we can really get a solid picture of what's in our data and what's reasonable to expect from model behavior. That's very different from training a single black-box model and checking a few test data assessment metrics. We're starting to learn how these models work, so we can make human judgment calls about their real-world performance.

What's more, you can use this information about interactions to boost the performance of linear models, such as a penalized logistic regression, by including these learned interactions as input features. If you want to stick with the most conservative model forms for your highest risk applications, you can use a GLM and likely get a boost in performance quality with this information about important interactions! As you can see, we can make all kinds of simple, interpretable observations about our XGBoost model through the use of decision tree surrogates. And we've collected loads of useful information for model documentation and other purposes along the way.

Shapley Value Explanations

The last post-hoc explanation tool to discuss before we close the chapter is Shapley values. In the brief **feature importance** discussion above, we mentioned that Shapley values can be used as a local feature attribution technique. In fact, Shapley values come with a host of mathematical guarantees that suggest that they are usually the best choice among for feature attribution and importance calculations. The research and open-source communities, led by Scott Lundberg at University of Washington and Microsoft Research, have developed a host of tools for generating and visualizing Shapley additive explanations (called SHAP values). These tools live in the `shap` Python package, and that's what we'll use in this section.

Remember that *local* feature attribution methods assign a value to each feature for each observation, quantifying how much that feature contributed to the predicted value that the observation received. In this section, we'll see how you can use SHAP values and the `shap` package to explain the behavior of your models. In the final section of this chapter, we'll examine some of the subtleties to Shapley-value based explanations, and the pitfalls they pose to practitioners.

Let's take a look at some code to generate SHAP values for our monotonic XGBoost model:

```
explainer = shap.TreeExplainer(model=model_constrained)
shap_values = explainer(train[features])
```

We're using the `shap` package's `TreeExplainer` class. This class can generate SHAP values for XGBoost, LightGBM, CatBoost, Pyspark, and most tree-based sklearn models. The `TreeExplainer` algorithms are introduced in Scott Lundberg's 2020 paper [From local explanations to global understanding with explainable AI for trees](#). They're a great example of the computational breakthroughs that have made the `shap` package so successful. If you need to generate SHAP values for a model not based on trees, take a look at the examples in the [shap package documentation](#) - there are multiple examples for tabular, text, and image data. But we caution that SHAP works best for trees in our opinion.

To begin, let's take a look at the SHAP values associated with the `PAY_0` feature in [Figure 6-10](#) below.

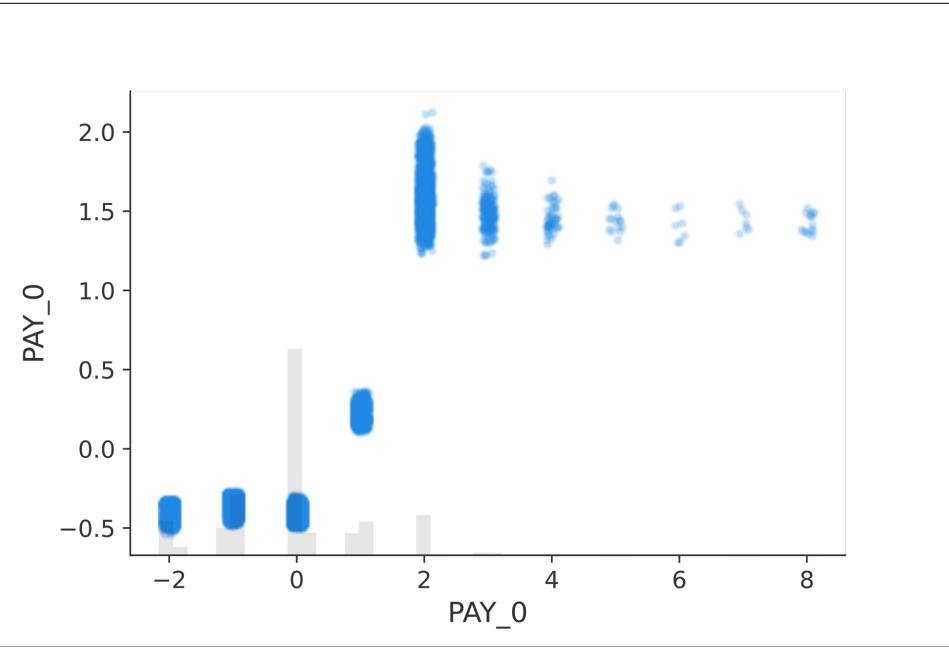


Figure 6-10. Dependence plot for the PAY_0 feature, showing the distribution of shap values in each bucket of feature values.

Each dot in [Figure 6-10](#) is one observation’s SHAP value for the PAY_0 feature, or contribution to the model prediction, with an x-coordinate given by the value of PAY_0. The scatter plot is superimposed over a histogram of feature values in the dataset, just like our partial dependence and ICE plots. In fact, this scatter plot can be directly compared to the partial dependence and ICE plots for PAY_0. In the shap scatter plot, we can see the whole range of feature attribution values within each bucket of PAY_0 values. Notice that the range is widest in the PAY_0 = 2 bucket — some observations with PAY_0 = 2 are penalized approximately half as much as others. This SHAP scatter plot is one of many summarizing plots included in the shap package. For a more thorough overview, take a look at the [examples in the documentation](#) as well as this chapter’s Jupyter notebook link:[examples](#).

As we saw in Chapter 2, we can construct a measure of overall *feature importance* by taking the mean of absolute SHAP values. Instead of standard horizontal bar chart of feature importance values, we can use shap’s plotting functionality to look at feature importance explicitly as an aggregation of local explanations:

```
shap.plots.beeswarm(shap_values.abs, color="shap_red", max_display=len(features))
```

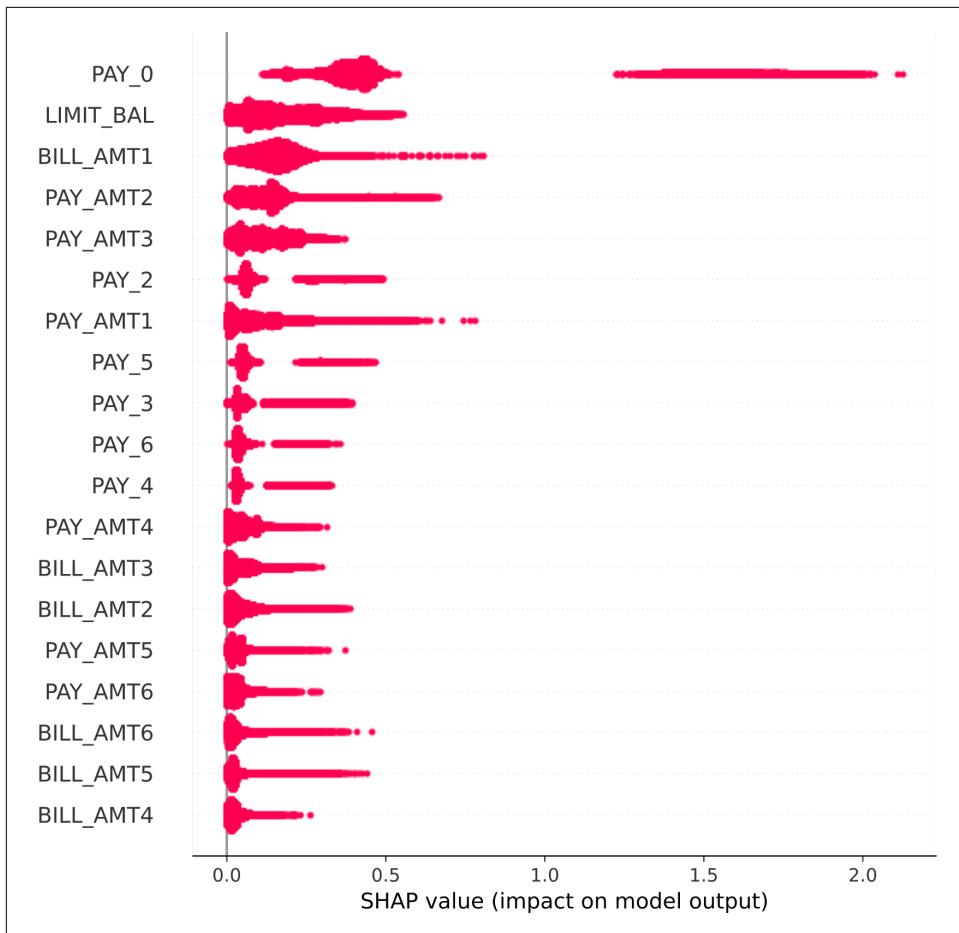


Figure 6-11. Feature importances, shown as the aggregation of individual observations' absolute shap values.

Figure 6-11 gives us an interesting perspective on feature importance. Notice that some features (PAY_0, PAY_AMT1) have a few dots exhibiting extreme SHAP values, whereas other features (LIMIT_BAL, PAY_AMT3) have a high feature importance because lots of individual observations have somewhat high absolute SHAP values. Put another way, local explanations allow us to distinguish between high-frequency, low-magnitude effects versus low-frequency, high-magnitude effects. This is important, because each of those high-magnitude, low frequency effects represent real people being impacted by our model!

Shapley-value-based explanations have seen widespread adoption due to their strong theoretical underpinning and a robust set of tools built around them. Since these quantities can be computed at an observation-by-observation level, they can likely be

used to generate adverse action notices or other turn-down reports (when used properly). Have a look at our code [examples](#), read some papers and documentation, and start generating SHAP values for your model! But please read the next section, where we discuss some of the implicit assumptions behind every SHAP computation.

Problems with Shapley Values

In the [concept refresher](#), we introduced the idea of a *background dataset* that underlies Shapley value calculations. Here's a useful way to think of background datasets: when you calculate a Shapley value (or in our case, a SHAP value) for an observation, you're answering the question "Why did this observation get this prediction *rather than some other prediction?*" The "other prediction" that observations are compared against is dictated by the choice of reference distribution! In the following code, we create two sets of SHAP values for the same observations - except in one instance we do not specify a reference distribution, and in the other we do:

```
explainer_tpd = shap.TreeExplainer(model=model_constrained,
                                     feature_perturbation='tree_path_dependent')
shap_values_tpd = explainer_tpd(train[features])

train['pred'] = model_constrained.predict(dtrain)
approved_applicants = train.loc[train['pred'] < 0.1]
explainer_approved = shap.TreeExplainer(model=model_constrained,
                                         data=approved_applicants[features],
                                         feature_perturbation='interventional')

shap_values_approved = explainer_approved(train[features])
```

By setting `feature_perturbation='tree_path_dependent'`, we're opting not to define a reference distribution at all. Next, we define an explainer for which we pass a reference distribution composed of training samples that received a probability of delinquency less than 10%. If the reference distribution is what we compare each observation against, then we would expect these two sets of SHAP values to be meaningfully different. After all, these questions are incredibly different: "Why did this observation get this prediction, *rather than the average prediction in the training data?*" versus "Why did this observation get this prediction, *rather than the predictions given to approved applicants?*" If you've worked in consumer finance, then you know the latter question is much more specifically meaningful than the former! Along with being more clearly meaningful, the latter question is much more aligned with US regulatory commentary on adverse action notices, or the mandated notices that must be sent to consumers who are denied credit.

The choice between `feature_perturbation='tree_path_dependent'` and `feature_perturbation='interventional'` might seem small. But it's not. Both choices come with assumptions and pros and cons. When you choose `feature_perturbation='tree_path_dependent'` you are selecting to generate explanations that are

more true to the training data than true to your specific model. And you are assuming that no complex input feature correlations will throw off your feature attribution calculations and that you can truly calculate individual feature contributions accurately. The first assumption is addressed by using a sane number of features, where none are too correlated. The second is usually a safe assumption with decision trees, but not with other ML models. When you choose `feature_perturbation='interventional'`, you are deciding to generate explanations that are more aligned with your specific model and that no strange out-of-range values in your background data are going to lead to unrealistic feature contributions.

For some observations, you can see large differences in the SHAP values under these two explanations. In Figure 6-12 below, we show two sets of SHAP values for the same observation - one calculated without a reference distribution, and one calculated against a reference of *approved* applicants.

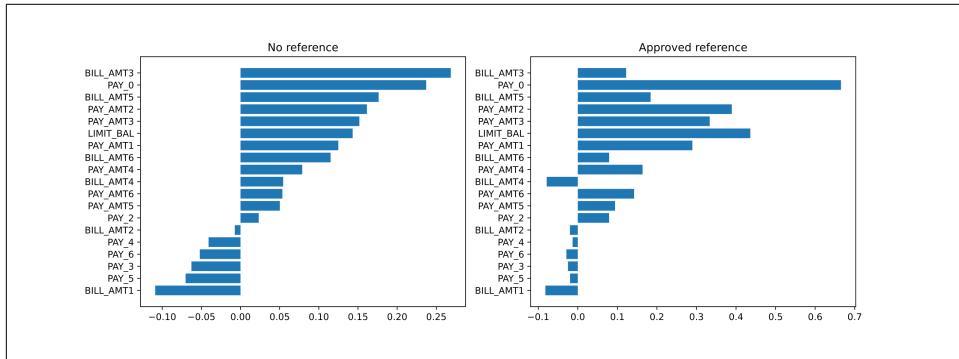


Figure 6-12. shap values for the same observation, generate with and without a reference distribution consisting of observations with an assigned probability of delinquency less than 10%.

Imagine adverse action notices sent out on the basis of these two explanations. With no reference distribution, the top four features that contributed to a larger probability of delinquency are BILL_AMT3, PAY_0, BILL_AMT5, and PAY_AMT2. But if we specify a context-specific reference distribution, these top four features are PAY_0, LIMIT_BAL, PAY_AMT2, and PAY_AMT3. Which explanation is correct? That depends on the question that you want answered. In a credit lending context, where *recourse* (the ability to appeal a model decision) is a crucial part of trust and responsible deployment, then the second explanation is likely the more correct choice. A denied applicant wants to know how to change their credit profile in order to get approved for credit in the future - that's the question posed by our approved applicant reference distribution. However, we need to exercise some caution even when using a meaningful and context-specific reference distribution. The recourse question, "what should I change [about my credit profile] in order to receive a favorable outcome in the future?" is

fundamentally a *causal* question. And we're not working with causal models. To quote the creator of the `shap` package, Scott Lundberg, “**Be careful when interpreting predictive models in search of causal insights**”. He goes on to say:

Predictive machine learning models like XGBoost become even more powerful when paired with interpretability tools like SHAP. These tools identify the most informative relationships between the input features and the predicted outcome, which is useful for explaining what the model is doing, getting stakeholder buy-in, and diagnosing potential problems. It is tempting to take this analysis one step further and assume that interpretation tools can also identify what features decision makers should manipulate if they want to change outcomes in the future. However, [...] using predictive models to guide this kind of policy choice can often be misleading.

For all of their mathematical guarantees and ease-of-use, Shapley-value-based explanations are not a magic wand. Instead, they are yet another explainability tool in your toolbox for explaining interpretable models. We have to combine our post-hoc explainability techniques with intrinsically interpretable model architectures such as GLMs, GAMs, or tightly-constrained XGBoost, to achieve true interpretability. And we have to stay humble and remember that ML is all about correlation, and not causation.

To conclude this chapter, let's return to the `PAY_0` feature, and compare how each of the five models we built treat this feature. Remember that `PAY_0` represents repayment status, where higher values correspond to a greater delay in repayment. Obviously, higher values should correspond to a greater risk of delinquency. However, the training data we used is sparse for higher values of the feature, so we have only a few observations more than one months delay. With that in mind, let's examine five partial dependence and ICE plots for each models' treatment of this feature in [Figure 6-13](#). Think to yourself, “which of these models would I trust the most with a billion-dollar lending portfolio?”

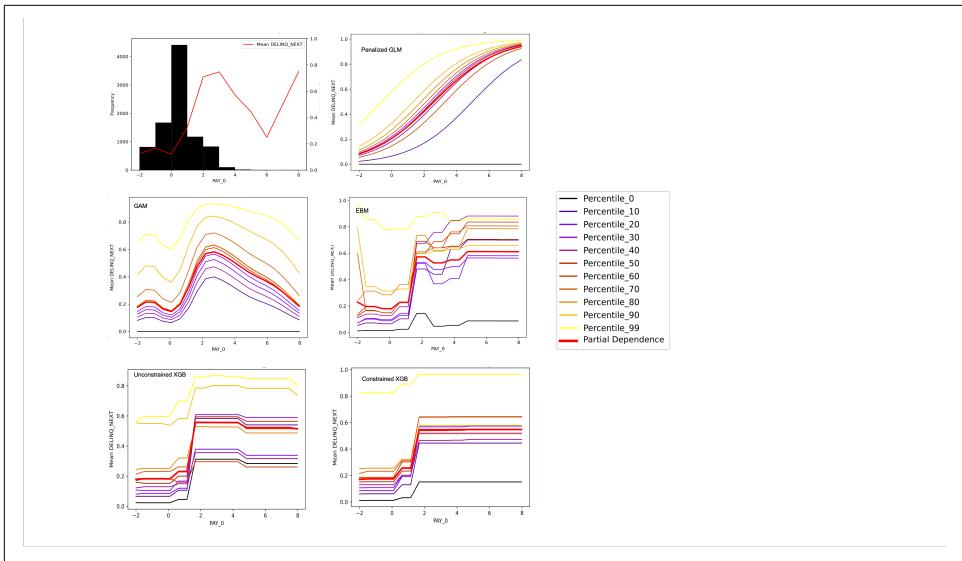


Figure 6-13. partial dependence and ICE plots for the five models trained in this chapter.

Three of our models show a response to this spurious dip in the mean target value in the sparse region of the feature space: the GAM, EBM, and unconstrained XGBoost. The GLM and constrained XGBoost model were forced to ignore this phenomenon. Since the GAM and EBM are additive models, we know that the partial dependence and ICE plots are truly representative of their treatment of this feature. The unconstrained XGBoost model is so full of feature interactions that we cannot be so sure. But the partial dependence does track with ICE, so it's probably a good indicator of true model behavior. We'd say it's a choice between the penalized GLM and the constrained XGBoost model. Which model would you pick if your job was on the line?

By using these interpretable models and post-hoc explainers, you can make a much more deliberate choice than in the traditional black-box ML workflow. Remember, if we were choosing by pure performance, we'd pick a model that treats our most important feature in a somewhat silly way! The story out of this deep dive into explainability in practice is this: first, understand which relationships between feature and target are truly meaningful, and which are noise. Second, if you need to be able to explain your models' behavior, and you probably do, choose a model architecture that is intrinsically interpretable. That way you'll have the model and the explainers to double check each other. Third, force your model to obey reality with constraints. People are still smarter than computers! Finally, examine your trained model with a diverse set of *post-hoc* explainability techniques such as partial dependence and ICE plots, surrogate models, and SHAP values. Working this way, you can make informed and reasonable model selection choices, and not simply overfit potentially biased and inaccurate training data.

Resources

Code Examples:

- GLM, GAM, and EBM code example
- Monotonic XGBoost and XAI code example

Reference:

- *Elements of Statistical Learning* (Chapters 3, 4, 9, and 10)

Open Source Code:

- **Interpretable Models:**

- arules
- causalml
- elasticnet
- gam
- glmnet
- H2O-3:
 - Monotonic gradient boosting machine
 - Sparse principal components
- imodels
- quantreg
- rpart
- RuleFit
- Rudin Group code
- sklearn-expertsys
- skope-rules
- tensorflow/lattice

- **Explainable AI:**

- ALEPlot
- Alibi
- anchor
- DiCE
- ICEbox

- `iml`
- `lime`
- `Model Oriented`
- `pdp`
- `shapFlex`
- `vip`

Debugging a PyTorch Image Classifier

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Even in the early glory days of deep learning (DL), researchers started to notice some **intriguing properties** of their new deep networks. The fact that a good model with high generalization performance could also be easily fooled by adversarial examples is both confusing and counter-intuitive. Similar questions were raised by authors in the seminal paper *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images* when they questioned how it was possible for a deep neural network to classify images as familiar objects even though they were totally unrecognizable to human eyes? If it wasn’t understood already, it’s become clear that like all other machine learning (ML) systems, DL models must be debugged and remediated, especially for use in high-risk scenarios. In Chapter 7, we trained a pneumonia image classifier and used various post-hoc explanation technique to summarize the results. We also touched upon the connection between DL explainability techniques and debugging. In this chapter we will pick up where we left off in Chapter 7 and use various debugging techniques on the trained model to ensure it is robust and reliable enough to be deployed.

DL represents the state-of-the-art in much of the ML research space today. However, its exceptional complexity also makes it harder to test and debug, which makes real-world adoption more risky. All software, even DL, has bugs and they need to be squashed before deployment. Chapter 9 starts with a concept refresher then focuses on model debugging techniques for DL models using our example pneumonia classifier. We'll start by discussing data quality and leakage issues in DL systems and why it is important to address them in the very beginning of a project. We'll then explore some software testing methods and why software quality assurance (QA) is an essential component of debugging DL pipelines. We'll also perform DL sensitivity analysis approaches, including testing the model on a dataset from a different distribution and applying adversarial attacks. We'll close the chapter by addressing our own data quality and leakage issues, discussing interesting new debugging tools for DL, and addressing the results of our own adversarial testing.

Concept Refresher: Debugging Deep Learning

In Chapter 8, we highlighted the importance of model debugging beyond traditional model assessment to increase trust in model performance. The core idea in this chapter remains the same, albeit for DL models. Recalling our image classifier from Chapter 7, trained to diagnose pneumonia in chest X-Ray images, we concluded that we could not entirely rely on the post-hoc explanation techniques we applied, especially in high-risk applications. However, those explanation techniques did seem to show some promise in helping us debug our model. In this chapter, we'll begin where we left off in Chapter 7. Remember we used Pytorch for training and evaluating the model, and we'll debug that very model in this chapter to demonstrate debugging for DL models. To get us started, the chapter concept refresher dives into data quality and data leaks, software testing methods, and adapting the broad concept of sensitivity analysis to DL. Just like in more traditional ML approaches, any bug we find with those techniques should be fixed, and the concept refresher will touch on the basics of remediation. It is also important to note that while the techniques introduced in this chapter apply most directly to computer vision models, the ideas can often be used in domains outside of computer vision.

- **Data Quality:** Image data can have any number of data quality issues. **Pervasive erroneous labels** in many of the datasets used to pre-train large computer vision models is one known issue. DL systems still require large amounts of labeled data, and are mostly reliant on fallible human judgment and labor to create those labels. Alignment, or making sure all the images in a training set have consistent perspectives, boundaries, and contents is another. Think about how difficult it is to align a set of chest X-rays from different X-ray machines on differently-sized people so that each of the training images focuses on the same content — human lungs — without distracting, noisy information around the edges. Because the contents of the images we're trying to learn about can themselves move up and

down or side to side (translate), rotate, or be pictured at different sizes (or scales), we have to have otherwise aligned images in training data for a high-quality model. Images also have naturally occurring issues, like blur, obstruction, low brightness or contrast and more. The recent paper *Assessing Image Quality Issues for Real-World Problems* does a good job at summarizing many of these common image quality problems and presents some methodologies for addressing them.

- **Data Leaks:** Another serious issue is leaks between training, validation, and test datasets. Without careful tracking of metadata, it's all too easy to have the same individuals or examples across these partitions. Worse, we can have the same individual or example from training data in the validation or test data at an earlier point in time. These scenarios tend to result in overly optimistic assessments of performance and error, which is one of the last things we want in a high-risk ML deployment.
- **Software Testing:** DL tends to result in complex and opaque software artifacts. For example, a *100-trillion-parameter* model. Generally, ML systems are also notorious for failing silently. Unlike a traditional software system that crashes and explicitly lets the user know about a potential error or bug through well-tested exception mechanisms, a DL system could appear to train normally and generate numeric predictions for new data, all while suffering from implementation issues. On top of that, DL systems tend to be resource-intensive, and debugging them is time-consuming as retraining the system or scoring batches of data can take hours. DL systems also tend to rely on any number of third-party hardware or software components. None of this excuses us from testing. It's all the more reason to test DL properly — software QA is a must for any high-risk DL system.
- **Sensitivity Analysis:** Sensitivity analysis in deep learning always boils down to changing data and seeing how a model responds. Unfortunately, there are just any number of ways images, and sets of images, can change when applying sensitivity analysis to DL. Interesting changes to images from a debugging standpoint can be visible or invisible to humans and they can be natural or made by adversarial methods. Entire sets of images used to train and test DL models can also be problematic. For example the populations within a set of images can change over time. Known as *subpopulation shift*, the characteristics of similar objects or individuals in images can change over time, and new subpopulations can be encountered in new data. The entire distribution of a set of images can change once a system is deployed too. For example, applying our pneumonia classifier on a new set of images taken from a new location and on a wholly different population of individuals. Another classic sensitivity analysis approach is to perturb the labels of training data. If our model performs just as well on randomly-shuffled labels, or the same features appear important for shuffled labels, that's not a good sign. Finally, we can perturb our model itself. One test for *underspecification* — or

when models work well in test data but not the real world — is to perturb structurally meaningless hyperparameters, like random seeds and number of GPUs used to train the system. If these abstract hyperparameters have a big effect on model performance, we’re still too focused on our particular training, validation, and tests sets.



Another important type of debugging that we would normally attempt is segmented error analysis, to understand how our model performs in terms of quality, stability, or over- and under-fitting across important segments in our data. Our X-ray images are not labeled with much additional information that would allow for segmentation, but understanding how a model performs across segments in data is crucial. Average or overall performance measures can hide underspecification and bias issues. If possible, we should always break our data down by segments and check for any potential issues on a segment-by-segment basis.

- **Remediation:** As with all ML, more and better data is the primary remediation method for DL too. Automated approaches that augment data with distorted images, like [albumentations](#) may be a workable solution in many settings for generating more training and test data. Once we feel confident about our data, basic QA approaches, like unit and integration testing and exception handling can help to catch many bugs before they result in sub-optimal real-world performance. Special tools like the Weights and Biases [experiment tracker](#) can enable better insight into our model training, helping to identify any hidden software bugs. We can also make our models more reliable and robust from the ML perspective using many different approaches. We can apply regularization, constraints based on human domain knowledge, or [Robust ML](#) approaches designed to defend against adversarial manipulation.

Debugging DL can be particularly difficult for all the reasons discussed in the concept refresher and more. However, we hope this chapter provides practical ideas for finding and fixing bugs. Let’s dive into our Chapter 9 case. We’ll be on the lookout for data quality issues, data leaks, software bugs, and undue sensitivity in our model in the sections below. We’ll find plenty of issues, and try to fix them.

Debugging a PyTorch Image Classifier

As we’ll discuss below, we ended up manually cropping our chest X-rays to address serious alignment problems. We found a data leak in our validation scheme, and we’ll cover how we found and fixed that. We’ll go over how to apply an experiment tracker and the results we saw. We’ll try some standard adversarial attacks, and discuss what

we can do with those results to make a more robust model. We'll also apply our model to an entirely new test set and go over performance on new populations. In the first sections below, we'll address how we found our bugs and some general techniques we might all find helpful for identifying issues in DL pipelines. We'll then discuss how we fixed our bugs, and some general bug remediation approaches for DL.

Data Quality and Leaks

As also highlighted in Chapter 7, the [pneumonia x-ray dataset](#) used in this case study comes with its own set of challenges. It has a skewed target class distribution. (This means there are more images belonging to the pneumonia class than the normal class.) The validation set is likely too small to draw meaningful conclusions. And there are markings on the images in the form of inlaid text or tokens. Typically, every hospital or department has specific style preferences for the X-Rays generated by their machines. When carefully examining the images, we observe a lot of unwanted markings, probes, and other noise as shown in Figure 7-1. Known as *shortcut learning*, these markers can become the focus of the DL learning process if we're not extremely diligent.

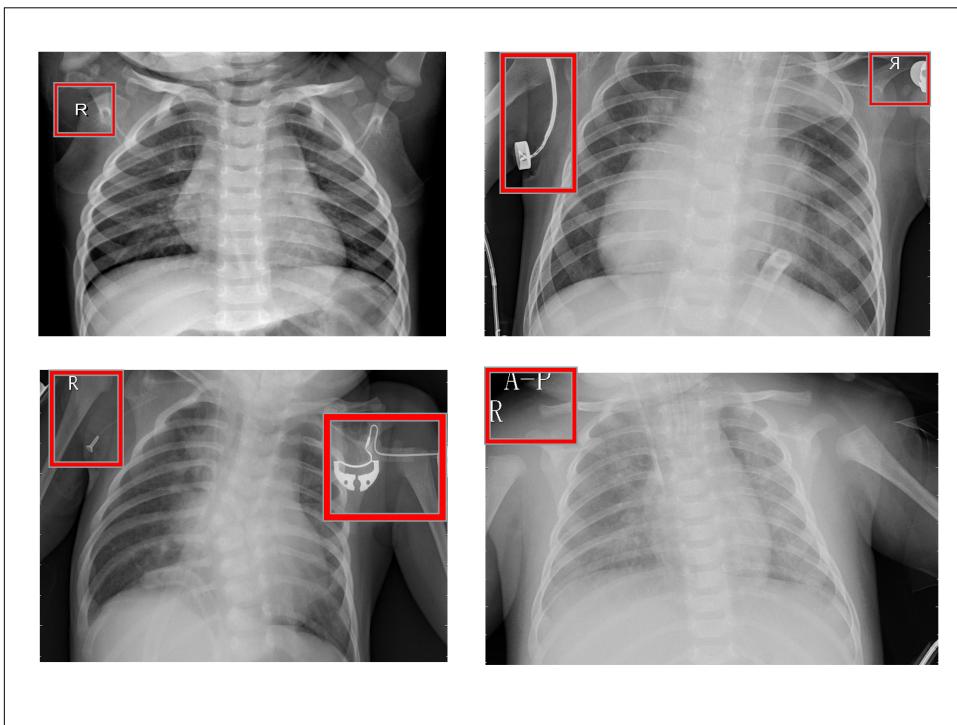


Figure 7-1. Images with unwanted inlaid text and markings.

We also uncovered a data leak. Simply put, a data leak occurs when information from validation or test data is available to the model during training time. A model trained on such data will exhibit optimistic performance on the test set but may perform poorly in the real world. Data leakage in DL can occur for many reasons, including:

- **Random splitting of data partitions:** This is the most common cause of leakage and occurs when samples representing the same individual are found in the validation or test data sets, and also appear in the training set. In this case, because of multiple images from the same individual in training data, a simple random split between training data partitions can result in images from the same patient occurring in the training and validation or test sets.
- **Leakage due to data augmentation** - Data augmentation is often an integral part of DL pipelines, used to enhance both the representativeness and quantity of training data. However, if done improperly, augmentation can be a significant cause of data leaks. If we're not careful with data augmentation, new synthetic images generated from the same real image can end up in multiple datasets.
- **Leakage during Transfer Learning** - Transfer learning can sometimes be a source of leakage when the source and target datasets belong to the same domain. In one [study](#), authors examine ImageNet training examples that are highly influential on CIFAR-10 test examples. They find that these images are often identical copies of images from the target task, just with a higher resolution. When these pretrained models are used with the wrong datasets, the pretraining itself results in a very sneaky kind of data leakage.

In our use case, we discovered that the training set contains multiple images from the same patient. Even though all the images have unique names, we observed instances where a patient has more than one x-ray, as shown in Figure [Figure 7-2](#).

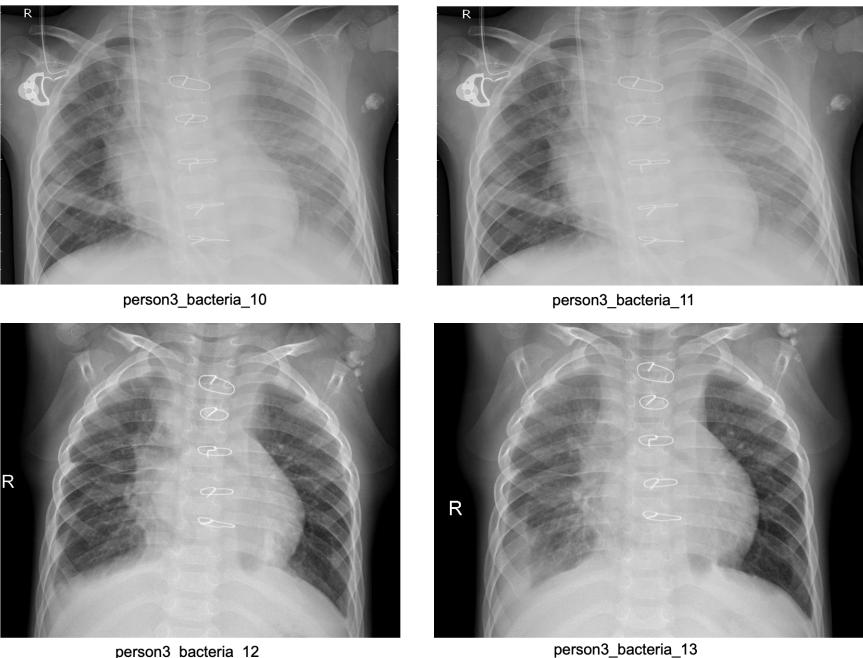


Figure 7-2. Multiple chest X-Ray Images from a single patient in the training set.

When images similar to [Figure 7-2](#), from a single patient, are sampled as part of the training set and as part of the validation or test set, it leads artificially high performance on the test set. This likely occurs because the network memorizes some specific elements from the x-ray for a single patient, and when it encounters similar data in the test set, it leans on that memorized data to make better decisions. In the real-world, the model can't depend on seeing the same people that are in its training data, and it may perform much worse than expected when faced with new individuals. Another data concern to keep an eye on is mislabeled samples. Since we are not radiologists, we cannot possibly pick a correctly labeled image from an incorrectly labeled image. This is just one place in a DL workflow to involve domain experts, and verify the correctness of the labels in our datasets.

Software Testing for Deep Learning

The tests specified in Chapter 3, namely unit, integration, functional, and chaos tests, can all be applied to DL systems, hopefully increasing our confidence that our pipeline code will run as expected in production. While software QA increases the chances are code mechanisms operate as intended, ML and math problems can still occur. DL systems are complex entities involving massive data and parameter sets. As such,

they also need to undergo additional ML-specific tests. Random attacks are a good starting point. Exposing the models to a large amount of random data can help catch a variety of software and ML problems. Benchmarking is another helpful practice discussed in numerous instances in Chapter 3. By comparing a model to benchmarks, we can conduct a sanity check on the model’s performance. Benchmarks can help us track system improvements over time in a systematic way. If our model doesn’t perform better than a simple benchmark model, or its performance is decreasing relative to recent benchmarks, that’s a sign to revisit our model pipeline.

The paper *A Comprehensive Study on Deep Learning Bug Characteristics* does an excellent job of compiling the most common software bugs in DL. The authors performed a detailed study of posts from Stack Overflow and bug fix commits from GitHub about the most popular DL libraries, including PyTorch. They concluded that data and logic bugs are the most severe bug types in DL software. QA software for DL is also becoming available to aide in detecting and rectifying bugs in DL systems. For instance, **DEBAR** is a technique that can detect numerical bugs in neural networks at the architecture level before training. Another technique **GRIST**, piggy-backs on the built-in gradient computation functionalities of DL infrastructures to expose numerical bugs. For testing NLP models specifically, **checklist** generates test cases, inspired by principles of functional testing in software engineering.

In our use case, we have to admit to not applying unit tests or random attacks as much as we should have. Our testing processes ended up being much more manual. In addition to wrestling with data leaks and alignment issues — a major cause of bugs in DL — we used informal benchmarks over the course of several months to observe and verify progress in our model’s performance. We also sanity checked our pipeline against the prominent bugs discussed in *A Comprehensive Study on Deep Learning Bug Characteristics*. We applied experiment tracking software too, which helped us visualize many complex aspects of our pipeline and feel more confident it was performing as expected. We’ll discuss the experiment tracker and other data and software fixes in more detail in the remediation section below.

Sensitivity Analysis for Deep Learning

We’ll use sensitivity analysis again to assess the effects of various perturbations on our model’s predictions. A common problem with ML systems is that while they perform exceptionally well in favorable circumstances, things get messy when they are subject to even minor changes in input data. Studies have repeatedly shown that even minor changes to input data distributions can **compromise the performance of state-of-the-art models** like DL systems. In this section, we’ll use sensitivity analysis as a means to evaluate our model’s robustness. Our best model will undergo a series of sensitivity tests for involving distribution shifts and adversarial attacks to ascertain if it can perform well in conditions different from which it was trained. We’ll also briefly cover a few other perturbation tricks throughout the chapter.

Domain and Subpopulation Shift Testing

Distribution shifts are a scenario when the training distribution differs substantially from the test distribution, or the distributions of data encountered once the system is deployed. These shifts can occur for various reasons and affect models that may have been trained and tested properly before deployment. Sometimes there is natural variation in data beyond our control. For instance, a pneumonia classifier created before the COVID-19 pandemic may show different results when tested on data after the pandemic. Perhaps the proportion of severe pneumonia cases during the pandemic increased considerably and a model trained prior to the pandemic might not have been exposed to many such cases in training data. Since distribution shift is so likely in our dynamic world, it is essential to detect it, measure it, and take corrective actions in a timely manner. While changes in data distributions are probably inevitable, and there may be multiple reasons why those changes occur. In this section, we'll first focus on domain shifts, i.e., when new data is from a different domain, which in this case would be another hospital. Further below, we'll highlight less dramatic — but still problematic — subpopulation shifts.

We trained our pneumonia classifier on a dataset of pediatric patients from [Guangzhou Women and Children's Medical Center](#) within one to five years of age. To check the robustness of the model to the dataset from a different distribution, we evaluate its performance on a dataset from another hospital and age group. Naturally, our classifier hasn't seen the new data, and its performance would indicate if it is fit for broader use. Doing well in this kind of test is difficult, and that is referred to as *out-of-distribution generalization*.

The new dataset comes from the [NIH Clinical Center](#) and is available through the [NIH download site](#). The images in the dataset belong to 15 different classes — 14 for common thoracic diseases, including Pneumonia, and 1 for “No findings,” where “No findings” means the 14 listed disease patterns are not found in the image. Each image in the dataset can have multiple labels. The dataset has been [extracted from the clinical PACS database](#) at the National Institutes of Health Clinical Center and consists of ~60% of all frontal chest x-rays in the hospital.

As mentioned above, the new dataset differs from the training data in several ways. First, unlike the training data, the new data has labels other than pneumonia. To take care of this difference, we manually extracted only the “Pneumonia” and “No Findings” images from the dataset and stored them as pneumonia and normal images. We assume that an image that doesn't report the 14 major thoracic diseases can be reasonably put in the normal category. Our new dataset is a subsample of the NIH dataset, and we have created it to contain almost balanced samples of pneumonia and normal cases. Again, this implicit assumption that half of screened patients have pneumonia may not hold, especially in real-world settings, but we want to test our best model obtained in Chapter 7 in distribution shift conditions, and this is the data we found. In Figure [Figure 7-3](#), we compare the chest x-rays from the two test sets,

visually representing the two different distributions. The lower set of images in the figure is sampled from a completely different distribution, while the images on the top are from the same distribution as the training set. While we don't expect a pneumonia classifier trained on images of children to work well on adults, we do want understand how poorly our system might perform under domain shift. We won't measure and document the limitations of our system, and know when it can and cannot be used. This is a good idea for all high-risk applications. In DL, it's often more of a visual exercise because each training data example is an image. In structured data, we might rely more of descriptive statistics to understand what data counts as out-of-distribution.

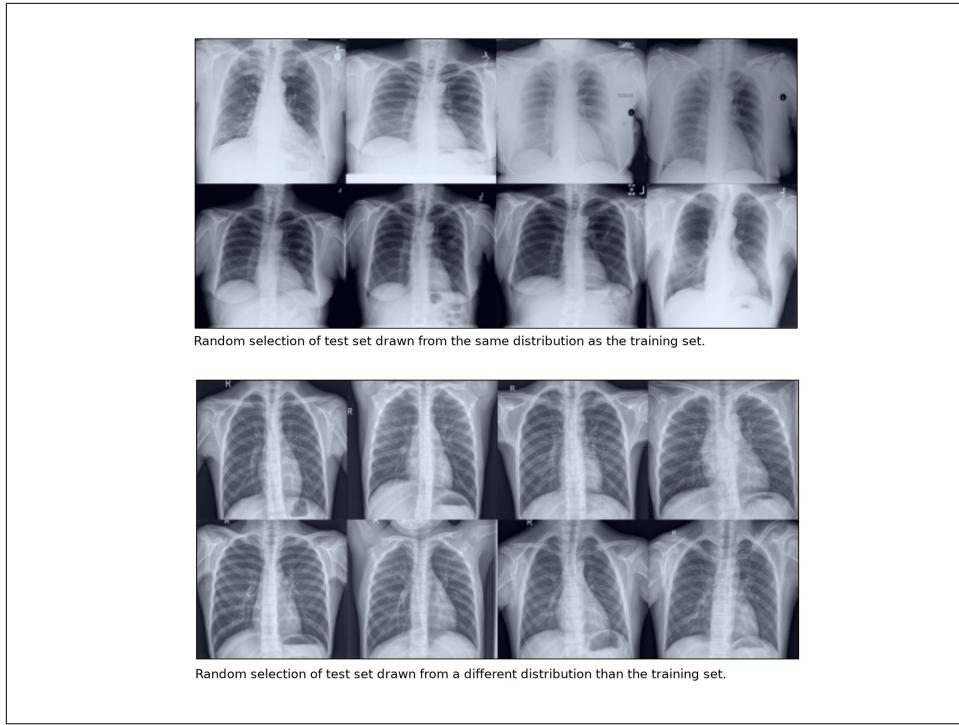


Figure 7-3. Comparison of X-Ray samples from two different distributions of data.

To our untrained eyes, both sets of images look similar. It is hard for us to differentiate between pneumonia and normal patient x-ray scans. The only difference we observed at first is that the images from the NIH dataset seem hazy and blurry compared to the other sample. A radiologist can, however, point out significant anatomical differences with ease. For instance, through reviewing literature we learned that the pediatric x-ray exhibits unfused growth plates in the upper arm that are not found in older patients. Since all the patients in our training data are children less than five years of age, their x-rays will likely exhibit this feature. If our model picks up on these

types of features, and somehow links them to the pneumonia label through shortcut learning or some other erroneous learning process, these spurious correlations will cause it to perform poorly on a new data where such a feature does not exist.

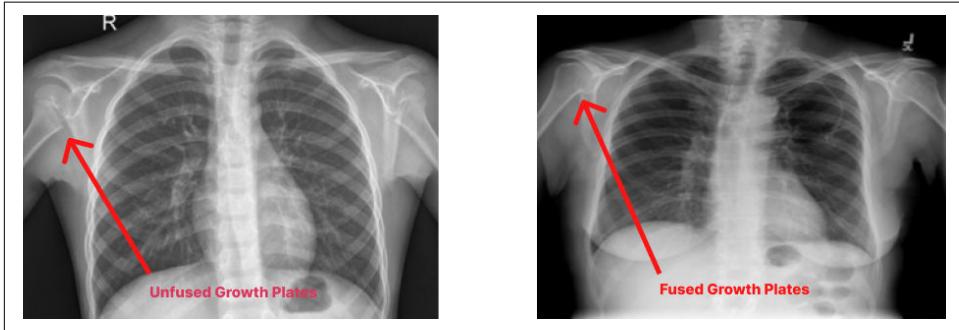


Figure 7-4. A pediatric X-Ray (Left) compared with that of an adult(Right).

Now for the moment of truth. We tested our best model on the test data from the new distribution and the results are not encouraging. We had our apprehensions going into this domain shift exercise, and they proved to be mostly true. Looking at the Figure [Figure 7-5](#) below, we can make some conclusions.

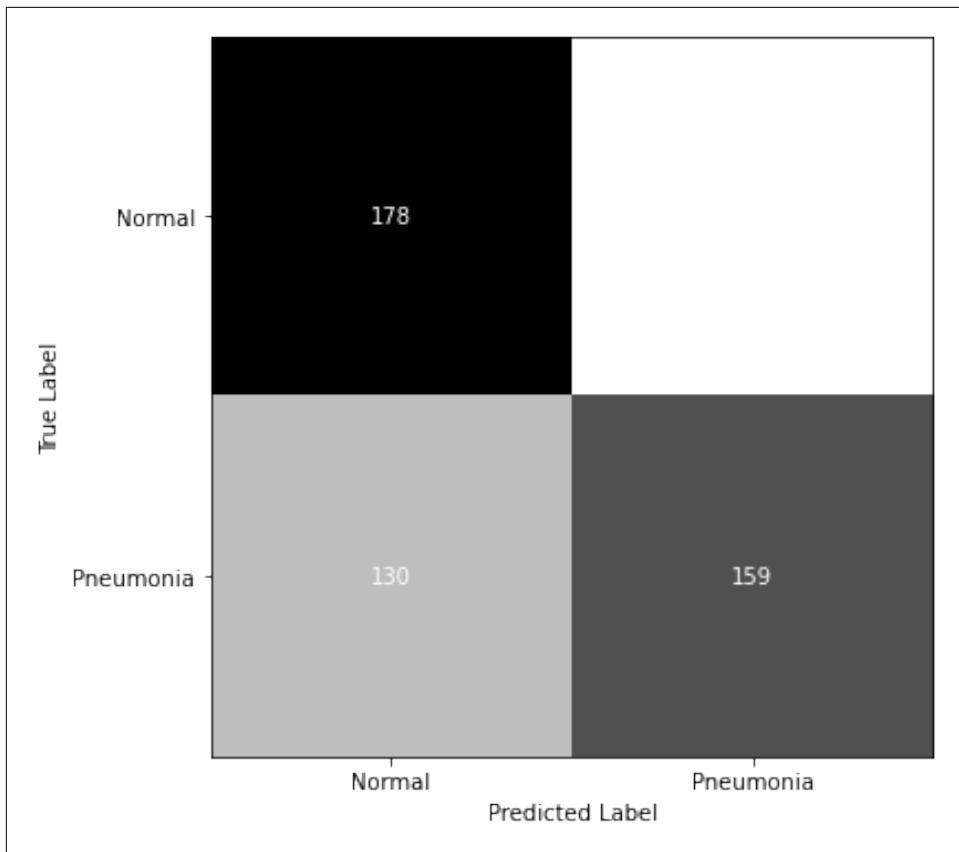


Figure 7-5. A confusion matrix showing the pneumonia classifier model performance on a test dataset from a different distribution.

The classifier incorrectly predicts the normal class for patients who actually had pneumonia fairly frequently. In the medical diagnostics context, false negatives — predicting patients with pneumonia are normal — are quite dangerous. If such a model is deployed in hospitals, it would have damaging consequences as sick patients may not receive correct treatments or timely treatments. [Table 7-1](#) below shows additional performance metrics for the classifier.

Table 7-1. Additional performance metrics on the test dataset from a different distribution.

Class	Precision	Recall	F1-Score	Support
Normal	0.58	0.64	0.61	280
Pneumonia	0.61	0.55	0.58	289

Could we have trained a better model? Did we have enough data? Did we manage the imbalance in the new dataset properly? Does our selection of samples in the new dataset represent a realistic domain shift? While we're not 100% sure of the answers to these questions, we did gain some clarity regarding our model's generalization capabilities, and how willing we are to deploy such models in high-risk scenarios. Any dreams we had that the generalist author team could train a DL classifier for pneumonia that works well beyond the training data have been dispensed with. We also think it's fair to reiterate just how difficult it is to train medical image classifiers.

Along with domain shifts, we also need to consider a less drastic type of data drift that can affect our classifier. Subpopulation shift occurs when we have the same population in new data, but with a different distribution. For example, we could encounter slightly older or younger children, a different proportion of pediatric pneumonia cases, or a different demographic group of children with slightly different physical characteristics. The approaches described in *BREEDS: Benchmarks for Sub-population Shift* focus on the latter case, where certain *breeds* of objects are left out of benchmark datasets, and hence not observed during training. By removing certain subpopulations from popular benchmark datasets the authors were able to identify and mitigate to some extent the effects of encountering new subpopulations. The same group of researchers also develops tools to implement the findings of their research on *robustness*. In addition to supporting tools for recreating the breeds benchmarks, the *robustness* package also supports various model training, adversarial training, and input manipulation functionality.

It's important to be clear-eyed about the challenges of ML and DL in high-risk scenarios. Training an accurate and robust medical image classifier today still requires large amounts of carefully labeled data, incorporation of specialized human domain knowledge, cutting-edge ML, and rigorous testing. Moreover, as the authors of *Safe and Reliable Machine Learning* aptly point out, it is basically impossible to know all the deployment environments during training time. Instead, we should strive to shift our workflows to proactive approaches that emphasize the creation of models explicitly protected against problematic shifts that are likely to occur. Next we'll explore adversarial example attacks, which help us understand both instability and security vulnerabilities in our models. Once we find adversarial examples, they can help us be proactive in training more robust DL systems.

Adversarial Example Attacks

We introduced adversarial examples in Chapter 8 with respect to tabular datasets. Recall that adversarial examples are strange instances of input data that cause surprising changes in model output. In this section, we'll discuss them in terms of our DL pneumonia classifier. More specifically, we'll attempt to determine if our classifier is capable of handling adversarial example attacks. Adversarial inputs are created by

adding a small but carefully crafted amount of noise to existing data. This noise, though often imperceptible to humans, can drastically change a model's predictions. The idea of using adversarial examples for better DL models rose to prominence in *Explaining and Harnessing Adversarial Examples*, where the authors showed how easy it is to fool contemporary DL systems for computer vision, and how tricky adversarial examples can be reincorporated into model training to create more robust systems. Since then, several studies focusing on safety-critical applications like **facial recognition** and **road sign classification**, have been conducted to showcase the effectiveness of these attacks. A great deal of subsequent **robust ML** research has concentrated on countermeasures and robustness against adversarial examples.

One of the most popular ways to create adversarial examples for DL systems is the fast gradient sign method (FGSM). Unlike the trees we work with in Chapter 8, neural networks are often differentiable. This means we can use gradient information to construct adversarial examples based on the network's underlying error surface. FGSM performs something akin to the converse of gradient descent. In gradient descent, we use the gradient of the model's error function with respect to the model's *weights* to learn how to change weights to *decrease* error. In FGSM, we use the gradient of the model's error function with respect to the *inputs* to learn how to change inputs to *increase* error. In the end, FGSM provides us with a *an* image, that often looks like static, where each pixel in that image is designed to push the model's error function higher. We use a tuning parameter, *epsilon*, to control the magnitude of the pixel intensity in the adversarial example. In general, the larger epsilon is, the worse error we can expect from the adversarial example. We tend to keep epsilon small, because the network usually just adds up all the small perturbations, affecting a large change in the model's outcome. As in linear models, small changes to each pixel (input) will typically add up to large changes in system outputs. We have to point out the irony, also highlighted by other authors, that the cheap and effective FGSM methods relies on DL systems mostly behaving like giant linear models.

A well-known example of the FGSM attack from *Explaining and Harnessing Adversarial Examples* shows a model first recognizes an image of a panda bear as a panda bear. Then FGSM is applied to create a perturbed, but visually identical, image of a panda bear. The network then classifies that image as a gibbon, or type of primate. While several packages like **cleverHans**, **foolbox** and **adversarial-robustness-toolbox** are available for creating adversarial examples, we manually implemented the FGSM attack on our fine-tuned pneumonia classifier based on the example given in the official PyTorch documentation. We then attack our existing fine-tuned model and generate adversarial images by perturbing samples from the test set, as shown in Figure [Figure 7-6](#). Of course, we're not trying to turn pandas into gibbons. We're trying to understand how robust our pneumonia classifier is to nearly imperceptible noise.

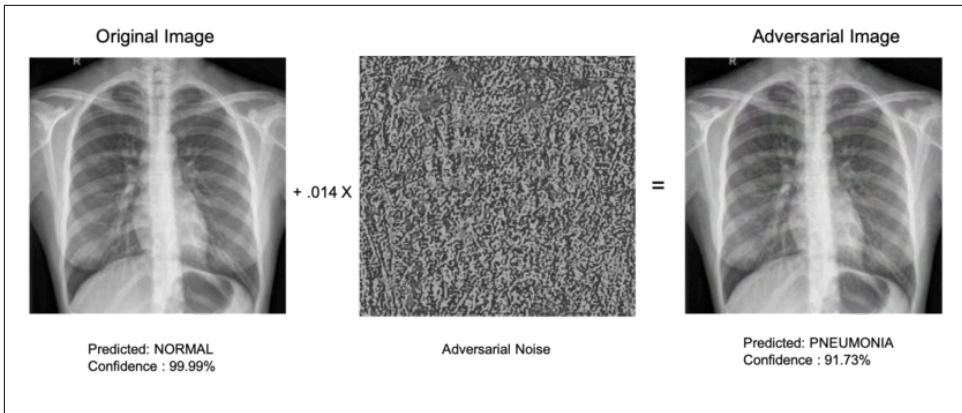


Figure 7-6. Invisible adversarial example attack shifts the prediction of a pneumonia classifier from Normal to Pneumonia.

The classifier which predicted an image in the normal class with a confidence of 99%, misclassified the FGSM-perturbed image as a pneumonia image. Note that the amount of noise is hardly perceptible. We also compare the value of epsilon against accuracy and plot them below. As expected, we can decrease our model's performance by increasing epsilon.

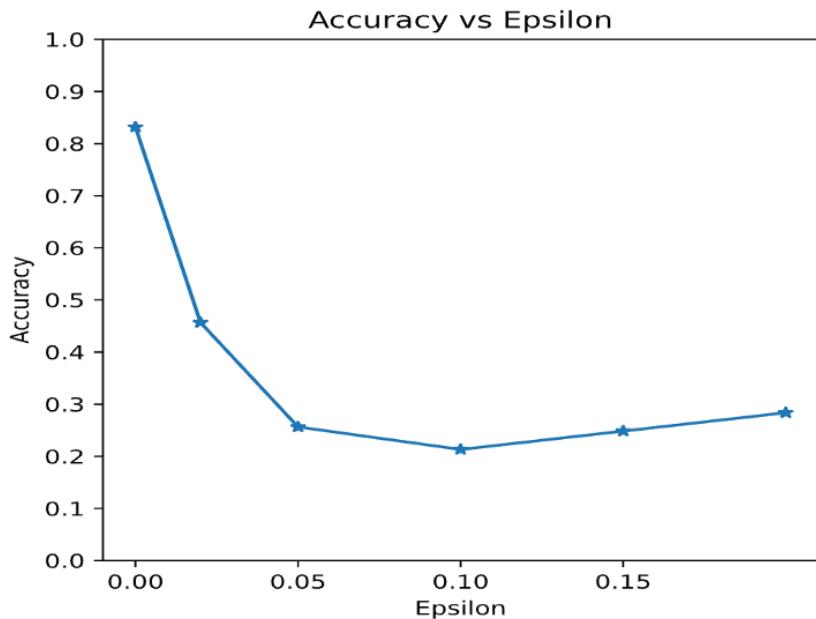


Figure 7-7. Accuracy vs Epsilon comparison for adversarial images.

Again, we're not doctors or radiologist. But how can we trust a system where invisible changes cause huge swings in predictions for such a high-stakes application? We have to be absolutely sure no noise has entered into our diagnostic images, either accidentally or placed there by a bad actor. We'd also like our model to be more robust to noise, just like we'd like it to be more robust to data drift. In the remediation section below, we'll outline some options for using adversarial examples in training to make DL systems more robust. For now, we'll highlight another perturbation and sensitivity analysis trick we can use to find other kinds of instability in DL models.

Perturbing Computational Hyperparameters

DL models require a large number of hyperparameters to be set correctly to find the best model for a problem. As highlighted in *Underspecification Presents Challenges for Credibility in Modern Machine Learning*, using standard assessment techniques to select hyperparameters tends to result in models that look great in test data, but that underperform in the real world. The *underspecification* paper puts forward a number of tests we can use to detect this problem. We touched on segmented error analysis above in several other chapters. Another way to test for underspecification is to perturb computational hyperparameters that have nothing to do with the structure of the

problem we are attempting to solve. The idea is that changing the random seed, or anything else that doesn't relate to the structure of the data or problem, say number of GPUs, number of machines, etc., should not change the model in any meaningful way. If it does, this indicates underspecification. If possible, try several different random seeds or distribution schemes (number of GPUs or machines) during training and be sure to test whether performance varies strongly due to these changes. The best mitigation for underspecification is to constrain models with additional human domain expertise. We'll discuss a few ways to do this in the next section on remediation.

Remediation

Usually when we find bugs, we try to fix them. This section will focus on fixing our DL model's bugs, and discuss some general approaches to remediating issues in DL pipelines. As usual, most of our worst issues arose from data quality. We spent a lot of time sorting out a data leak and manually cropping images to fix alignment problems. From there we analyzed our pipeline using a new profiling tool to find and fix any obvious software bugs. We also applied L2 regularization and some basic adversarial training techniques to increase the robustness of our model. We'll be providing some details on how all this done in the sections below, and we'll also highlight a few other popular remediation tactics for DL.

Data Fixes

In terms of data fixes, first recall that in Chapter 7 we addressed a data imbalance issue by carefully augmenting images. We also had to fix the data leak we uncovered, then deal with problems in image alignment. After these time-consuming manual approaches, we were able to apply a double fine tuning training approach that did noticeably improve our model's performance.

To ensure there is no leakage between individuals in different datasets, we manually extended the validation dataset by transferring unique images from the training set to the validation set. We augmented the remaining training set images using the transformations available in PyTorch, paying close attention to domain constraints relating to asymmetrical images. (Lung images are not laterally symmetrical, so we could not use augmentation approaches that resulted in lateral symmetry.) This eliminated the data leak. We could also have split our datasets by patient — ensuring all the x-rays that belong to the same patient are in the same data partition.



Partitioning data into training, validation, and test sets after augmentation is a common source of data leaks in DL pipelines.

The next fix we tried was manually cropping some of the x-rays with image manipulation software. While PyTorch has transformations that can help in center cropping of the x-ray images, they didn't do a great job on our data. So we bit the bullet, and cropped hundreds of images ourselves. In each case, we sought to preserve the lungs' portion of the x-ray images, get rid of the unwanted artifacts around the edges, and preserve scale across all images as much as possible. Figure [Figure 7-8](#) shows a random collection of images from the cropped dataset. We were also vigilant about not re-introducing data leaks while cropping, and made every effort to keep cropped images in their correct data partition.

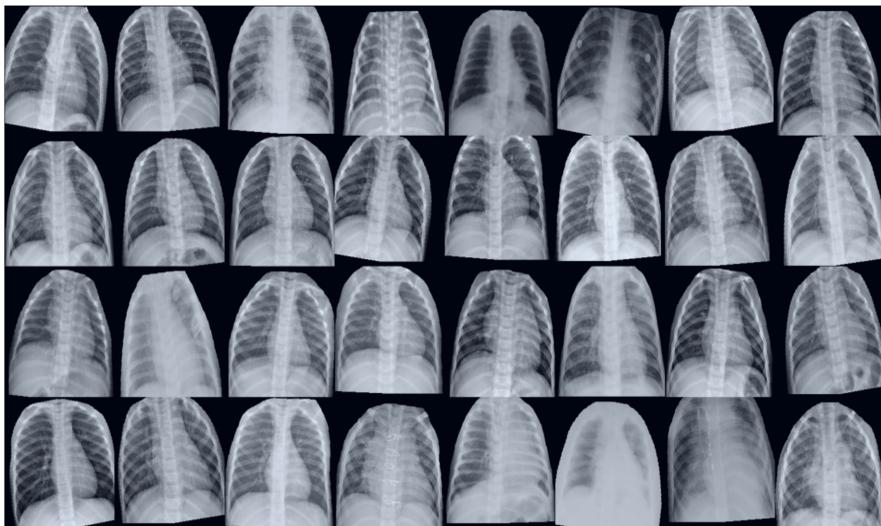


Figure 7-8. Manually-cropped x-ray images.

The major advantage of going through the laborious process of manual imaging cropping is to create another dataset that can be used for a two-stage transfer learning process. As also explained in Chapter 7, we use a pre-trained DenseNet-121 for transfer learning. However, the source data on which this architecture is trained varies significantly from our target domain. As such, we follow a process where we first fine tune the model on the augmented and leak-free dataset and then perform another fine tuning of the resultant model only on the cropped dataset. Figure [Figure 7-9](#) shows the performance comparison on the validation set after the second fine tuning stage.

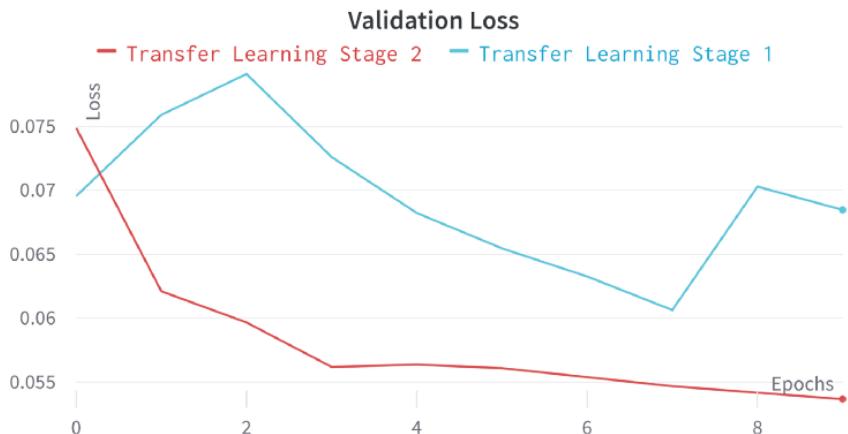


Figure 7-9. Performance comparison on the validation set for double fine tuning. (Lower is better.)

Figure Figure 7-10 shows the Test set loss performance after the second transfer learning stage.



Figure 7-10. Performance comparison on the test set for double fine tuning. (Lower is better.)

Since the double fine tuned model exhibits better performance on the validation data as well as the hold-out test set, we choose it as our best model. It took a lot of manual effort to get here, **which is likely the reality for many DL projects**. In our research

into fixing our data problems, we ran into the [albumentations](#) library that looks great for augmentations and the [label-errors](#) project that provides tools for fixing some common image problems. While we had to revert to manual fixes, these packages do seem helpful in general. After the long fight for clean data, and finding a fine tuning process that worked well for that data, it's time to double check our code.

Software Fixes

Since DL pipelines involve multiple components, there many components to test and we have to consider their integration as well. In the end, we may be left wondering did we select an appropriate model architecture? optimizer? batch size? loss function? activation function? learning rate? and on and on. To have any hope of answering these questions, we have to breakdown are software debugging into atomic steps that attempt to isolate issues one-by-one:

- **Sanity check our training device:** Before proceeding with training, ensure the model and data are always on the same device(CPU or GPU). It's a common practice in PyTorch to initialize a variable that holds the device we're training the network on (CPU or GPU).

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

- **Summarize network architecture:** Summarize the outputs from layers, gradients, and weights to ensure there is no mismatch.
- **Test network initialization:** Check the initial values of weights and hyperparameters. Consider whether they make sense and whether any anomalous values are easily visible. Experiment with different values if needed.
- **Confirm training settings on a mini-batch:** Overfit a small batch of data to check training settings. If successful, we can move on to a bigger batch. If not, we go back and debug our training loop and hyperparameters. The code below demonstrates overfitting a single batch in PyTorch.

```
single_batch = next(iter(train_loader))
for batch, (images, labels) in enumerate([single_batch] * no_of_epochs):

    # training loop
    # ...
```

- **Tune the (initial) learning rate:** A minimal learning rate will make the network converge very slowly but traverse the error surface more carefully. A high learning rate will do the opposite — jump around the error surface haphazardly, landing in some local minima. Choosing good learning rates is important and difficult. There are some open-source tools in Pytorch like [PyTorch learning rate finder](#) that can help determine an appropriate learning rate, as shown in the code

below. The paper *Cyclical Learning Rates for Training Neural Networks* discusses one way to choose DL learning rates in detail. These are just a few of the available options. If we're using a self-adjusting learning rate, we also have to remember we can't test that without training until we hit a realistic stopping criterion.

```
from torch_lr_finder import LRFinder

model = ...
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.1, weight_decay=1e-2)
lr_finder = LRFinder(model, optimizer, criterion, device="cuda")
lr_finder.range_test(trainloader, val_loader=val_loader, end_lr=1,
num_iter=100, step_mode="linear")
lr_finder.plot(log_lr=False)
lr_finder.reset()
```

- **Refine loss functions and optimizers:** Matching loss functions to the problem at hand is a must for usable ML results in general. With DL picking the best loss function is especially difficult, as there are so many options for both loss functions and optimizers. We also don't have convergence guarantees as we might with some linear models. In Pytorch, a common mistake is applying a softmax loss instead of the **Cross-Entropy loss**. For PyTorch, cross-entropy loss expects logit values, and passing probabilities to it as inputs will not give correct outputs. Try your loss and optimizer selection for a reasonable number of iterations, checking iteration plots and predictions to ensure the optimization process is progressing as expected.
- **Adjust regularization:** Contemporary DL systems usually require regularization to generalize well. However, there are many options (L1, L2, dropout, input dropout, etc.) and it's not impossible to go overboard. Too much regularization can prevent a network from converging, and we don't want that either. It takes a bit of experimentation to pick the right amount and type of regularization.
- **Test drive the network:** It's no fun to start a big training job just to find out it diverged somewhere along the way, or failed to yield good results after burning many chip cycles. If at all possible, train your network fairly deep into its optimization process and check that things are progressing nicely before performing the final long training run.
- **Improve reproducibility:** While stochastic gradient descent (SGD) and other randomness is built into much of contemporary DL, we have to have some baseline to work from. If for no other reason, we need to make sure we don't introduce new bugs into our pipelines. If our results are bouncing around too much, we can check our data splits, feature engineering, random seeds for different software libraries, and the placement of those seeds. (Sometimes we need to have seeds inside training loops.) There are also sometimes options for exact reproducibility that come at the expense of training time. It might make sense to suffer

through some very slow partial training runs to isolate reproducibility issues in our pipelines.

We must have performed these steps, identified errors, and re-tested hundreds of times — catching countless typos, errors, and logic issues along the way. Once we fix up our code, we want to keep it clean and have the most reproducible results possible. One way to do that efficiently is with newer experiment tracking tools like [Weights & Biases](#). These tools can really help in building better models faster by efficient dataset versioning and model management. Figure [Figure 7-11](#) shows how multiple experiments are being seamlessly tracked and visualized leading to less bugs and better reproducibility.



Figure 7-11. Tracking multiple experiments seamlessly with Weights & Biases.

While you can use the debugging steps above, unit tests, integration tests, and experiment trackers to identify and avoid commonly occurring bugs, another option is to try to avoid complex training code altogether. An excellent alternative to writing hundreds or thousands of lines of Python code is the [PyTorch Lightning](#) - an open-source Python library that provides a high-level interface for PyTorch. It manages all the low-level stuff, abstracting commonly repeated code, and enabling users to focus more on the problem domain than on engineering.



Low-code is not new. In fact, before the recent hype around deep learning, Python, and GPUs, a lot of the analytics industry had settled into using sophisticated low- or no-code modeling tools like Weka, RapidMiner, SAS Enterprise Miner and SPSS Modeler because they knew writing training code was difficult and error prone. By insisting on writing massive amounts of Python code, we're ignoring the wisdom of our predecessors to a certain extent.

While higher level libraries may feel like we're giving up control, the key to remember is they should *ideally* be pre-tested. Data scientists often forget that writing code is perhaps the fastest part of the development cycle. Every line of code we write has to be tested on it's own and in combination with other aspects of our pipeline. If low-code or higher-level libraries are tested well, this is a huge win. Now that we're feeling confident that our code pipeline is performing as expected and is not riddled with bugs, we'll shift to trying to fix mathematical stability issues in our network.

Sensitivity Fixes

Data has problems. Code has problems. We did our best to solve those in our DL pipeline. Now it's time to try to fix the math issues we found. The robustness problems we encountered in the sections above are not unique. They are some of the most well-known issues in DL. In the subsections below, we'll take inspiration from major studies about common robustness problems. We'll perform some actual remediation, and discuss several other options we could try.

Noise Injection

One of the most common causes of a network's poor generalization capability is overfitting. This is especially true for small datasets like we're using. Noise injection is an interesting option for customizing regularization and adding strong regularization to our pipelines. We decided to try it, intentionally corrupting our training data, as a way to add extra regularization to our training process. Adding noise to training samples can help to make the network more robust to input perturbations, and has effects similar to L1 or L2 regularization on model parameters. Adding noise to images can also result in data augmentation by creating artificial samples from the original dataset.



Injecting random noise in images is also known as *jitter* - a word that dates back decades and has its roots in signal processing. Injection of Gaussian noise is equivalent to L2 regularization.

We added a small amount of Gaussian noise to the training samples. Gaussian noise, also known as white noise, has a mean of zero, a standard deviation of one, and is

drawn from a normal distribution. We then re-train the model on the training data obtained after adding Gaussian noise and test the model on the new unseen dataset. We study the effect of noise on the model's robustness and whether or not we obtain any performance boost on hold-out data.

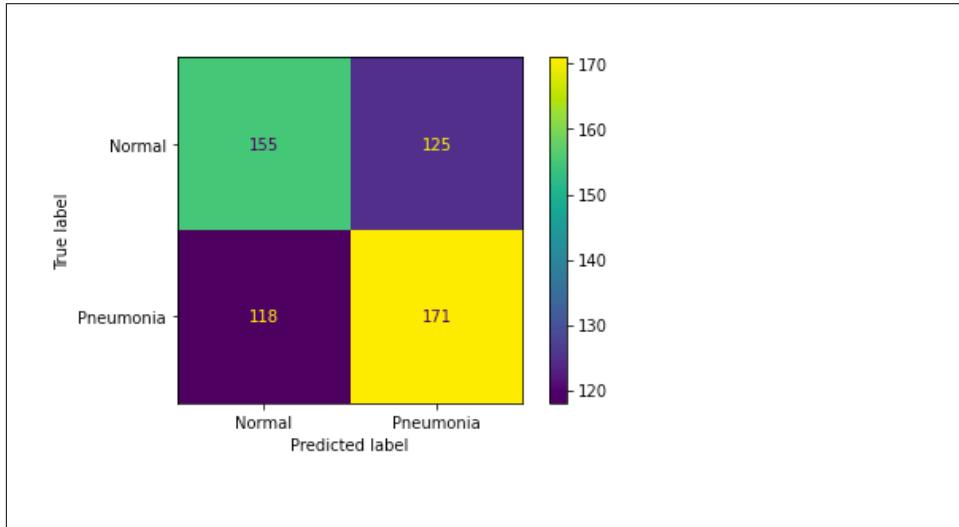


Figure 7-12. Performance comparison on the out of distribution data.

The regularized model performed a bit better on test data. However, as seen in Figure 7-12, the model only performed slightly better than random on our out-of-distribution dataset. So, noise injection did not make our model perform miraculously on out-of-distribution data. But, all things being equal, we'd like to deploy the most regularized model that also performed adequately on test data. Note that while we added noise only to the training samples, it can also be added to the weights, gradients and labels.

Additional Stability Fixes

We toyed around with many other stability fixes, but saw similar results to noise injection. Some helped a bit, but nothing fixed our out-of-distribution performance. However, that doesn't mean they didn't make our model better for some unseen data. Below, we'll go over more data augmentation options, learning with noisy labels, domain-based constraints, and robust ML approaches before closing out the chapter.

- **Automated data augmentation:** Another option for boosting robustness is exposing the network to a wider variety of data distributions during training. While it is not always possible to acquire new data, effective data augmentation is becoming somewhat turnkey in DL pipelines. [Albumentations](#) is a popular library for creating different types of augmented images for computer vision

tasks. Albumentations is easily compatible with popular DL frameworks such as PyTorch and Keras. [Augly](#) is another data augmentation library focused on creating more robust DL models available for audio, video, and text, in addition to images. The unique idea behind Augly is that it derives inspiration from real images on the internet and provides a suite of more than 100 augmentation options.

- **Learning with noisy labels:** For many different reasons, labels on images can be noisy or wrong. This can occur because of the volume of images required to label and then train a contemporary DL system is large, because of expenses associated with labels, because of the technical difficulties of labeling complex images, or other reasons. In reality, this means we are often training on noisy labels. At the most basic level, we can shuffle some small percentage of image labels in our training data and hope that makes our model more robust to label noise. Of course, there's always more we can do and learning on noisy labels is a busy area of DL research. The GitHub repo, [noisy_labels](#) lists a large number possible noisy label learning approaches and tools. Also, recall that in Chapter 7, we used label shuffling as a way to find robust features and sanity check explanation techniques. From our standpoint, using label-shuffling for explanation, feature selection, and sanity checking purposes may be its highest calling today.
- **Domain-based constraints:** To defeat underspecification, it's essential to incorporate domain information or prior knowledge into DL systems. One approach to integrating prior knowledge into DL models is to simulate training data based on domain knowledge, like equations that govern systems, and conduct unsupervised pretraining with this data. The weights learned during pre-training can then be used to initialize supervised training of the network. We can also employ monotonicity constraints, as is done with TensorFlow [lattice](#), to ensure modeled relationships between inputs and targets follow causal realities. Don't forget about the basics too. We need to match our loss functions to known target and error distributions. [Informed machine learning](#) is a paradigm that considers the source of knowledge, its representation, and its integration into ML pipelines. Check this out for even more domain constraint ideas.
- **Robust machine learning:** While it's a bit of a confusing name, *Robust Machine Learning* is the common phrase for the area of DL research that addresses adversarial manipulation of models. [Robust ML](#) is a community-run website that consolidates different defense strategies and provides various countermeasures and defenses, primarily for adversarial example attacks and data poisoning. While Robust ML is a wide area of study, some common methods include retraining on adversarial examples, gradient masking, and countermeasures for data poisoning:
 - **Retraining on adversarial examples:** Popular techniques include retraining on properly labeled adversarial examples, where those examples are found by methods like FGSM. (We tried this, but the results looked a lot like the noise

injection results.) This technique involves retraining the model with a combination of original data and adversarial examples, where the model should be more difficult to fool as it has already seen many adversarial examples. The paper “[Adversarial Examples Are Not Bugs They Are Features](#)” by the [Madry lab](#) offers more perspective into how we can understand adversarial examples in the light of identifying robust input features.

- **Gradient masking:** Gradient masking works by changing gradients so they aren’t useful to adversaries when creating adversarial examples. It turns out gradient masking isn’t actually a good defense and can be [circumvented easily](#) by motivated attackers. However, gradient masking is important to understand for red-teaming and testing purposes, as many other attacks were inspired by weaknesses in gradient masking. For example, the [foolbox](#) library has a good demonstration of *gradient substitution*, i.e., replacing the gradient of the original model with a smooth counterpart and building effective adversarial examples using that substituted gradient.
- **Data poisoning countermeasures:** There are a number of defenses for detecting and mitigating data poisoning. For example, the [Adversarial Robustness Toolbox](#) (ART) toolkit contains detection methods based on hidden unit activations, data provenance, and using spectral signatures. Respectively, the basic ideas are that triggering backdoors created by data poisoning should cause hidden units to activate in anomalous ways, as backdoors should be used in rare scenarios. That data provenance — developing a careful understanding and records about the handling of training data — can ensure it is not poisoned. And using principal components analysis (PCA) to find tell-tale signs of adversarial examples. To see an example for how ART works for detecting data poisoning, checkout the [activation defense demo](#).

As we can see, there are a lot of options to increase robustness in DL. If we’re most worried about robust performance on new data, noise injection, data augmentation, and noisy label techniques may be most helpful. If we have the ability to inject human domain knowledge, we should probably always do that. And if we’re worried about security and adversarial manipulation, we need to consider official robust ML methodologies. While there are some rules of thumb and logical ideas about when to apply which fix, we really have to try them all to find what works best for our data, model and application.

Conclusion

Even after all of this testing and debugging, we’re fairly certain we should not deploy this model. While none of the authors consider themselves DL experts, we do wonder what this says about the level of hype around DL. If the author team couldn’t get this model right after months of work, what does it take in reality to make a high-stakes

DL classifier work? We have access to nice GPUs and many years of experience in ML between us. That's not enough. Two obvious things missing from our approach are massive training data and access to domain experts. Next time we take on a high-risk application of DL, we'll make sure to have access to those kinds of resources. But that's not a project that a handful of data scientist can take on their own. A repeated lesson from this book is it takes more than a few data scientists to make high-risk projects work.

At a minimum, we'll need an entire supply chain to get properly labeled images and access to expensive domain experts. Even with those improved resources, we'd still need to perform the kind of testing described in Chapter 9. On the whole, our experiences with DL have left us with more questions than answers. How many DL systems are trained on smaller datasets and without human domain expertise? How many DL systems are deployed without the level of testing described in this chapter? In those cases, did the systems really not have bugs? Or maybe it was assumed they did not have bugs? For low-risk games and apps, these issues probably aren't a big deal. But for DL systems being used in medical diagnosis, law enforcement, security, immigration, and other high-risk problem domains, we hope the developers of those systems had access to better resources than us and put in serious testing effort.

Resources

- **Data Generation:**
 - [AugLy](#)
 - [faker](#)
- **Attacks and Debugging:**
 - [adversarial-robustness-toolbox](#)
 - [albumentations](#)
 - [cleverhans](#)
 - [checklist](#)
 - [counterfit](#)
 - [foolbox](#)
 - [robustness](#)
 - [tensorflow/model-analysis](#)
 - [TextAttack](#)
 - [TextFooler](#)
 - [torcheck](#)
 - [TorchDrift](#)

Testing and Remediating Bias with XGBoost

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

This chapter presents bias testing and remediation techniques for structured data. While [Chapter 4](#) addressed issues around bias from various perspectives, this chapter focuses on technical implementations of bias testing and remediation approaches. We’ll start off by training XGBoost on a variant of the credit card data. We’ll then test for bias by checking for differences in performance and outcomes across demographic groups. We’ll also try to identify any bias concerns at the individual observation level. Once we confirm the existence of measurable levels of bias in our model predictions, we’ll start trying to fix, or remediate, that bias. We employ pre-, in- and post-processing remediation methods that attempt to fix the training data, model, and outcomes, respectively. We’ll finish off the chapter by conducting bias-aware model selection that leaves us with a model that is both performant and minimally biased.

While we've been clear that technical tests and fixes for bias do not solve the problem of machine learning (ML) bias, they still play an important role in an effective overall bias mitigation or ML governance program. While fair scores from a model do not translate directly to fair outcomes in a deployed ML system — for any number of reasons — it's still better to have fair scores than not. We'd also argue it's one of the fundamental and obvious ethical obligations of practicing data scientists to test models that operate on people for bias. Another theme we've brought up before is that unknown risks are much harder to manage than known risks. When we know a system may present bias risks and harms, we can attempt to remediate that bias, monitor the system for bias, and apply many different socio-technical risk controls — like bug bounties or user interviews — to mitigate any potential bias.



[Chapter 8](#) focuses on bias testing and remediation for a fairly traditional classifier because this is where these topics are best understood, and because many complex artificial intelligence (AI) outcomes often boil down to a final binary decision that can be treated in the same way as a binary classifier. We highlight techniques for regression models throughout [Chapter 8](#) as well. See [Chapter 4](#) for ideas on how to manage bias in multinomial, unsupervised, or generative systems.

By the end of the chapter, readers should understand how to test a model for bias and then select a less biased model that also performs well. While we acknowledge there's no silver bullet tech fix for ML bias, a model that is more fair and more performant is a better option for high-risk applications than a model that hasn't been tested or remediated for bias. [Chapter 8](#) code examples are available on [Colab](#) and [GitHub](#).

Concept Refresher: Managing ML Bias

Before we dive into the [Chapter 8](#) case study, let's do a quick refresh of the applicable topics from [Chapter 4](#). The most important thing to emphasize from [Chapter 4](#) is that all ML systems are socio-technical, and the kind of purely technical testing we're focusing on in this chapter is not going to catch all the different bias issues that might arise from an ML system. The simple truth is that "fair" scores from a model, as measured on one or two datasets, give an entirely incomplete picture of the bias of the system. Other issues could arise from unrepresented users, accessibility problems, physical design mistakes, downstream misuse of the system, misinterpretation of results, and more.



Technical approaches to bias testing and mitigation must be combined with socio-technical approaches to adequately address potential bias harms. We can't ignore the demographic background of our own teams, the demographics of users or those represented in training and testing data, data science cultural issues (like entitled "rock stars" or "tech bros"), and highly developed legal standards, and also expect to address bias in ML models. This chapter focuses mostly on technical approaches. [Chapter 4](#) attempts to describe a broader socio-technical approach to managing bias in ML.

We must augment technical bias testing and remediation efforts with an overall commitment to having a diverse set of stakeholders involved in ML projects and adherence to a systematic approach to model development. We also need to talk to our users and abide by model governance that holds humans accountable for the decisions of the computer systems we implement and deploy. To be blunt, these kinds of socio-technical risk controls are likely more important and more effective than the technical controls we discuss below.

Nonetheless, we don't want to deploy blatantly biased systems, and if we can make the technology better, we should. Less biased ML systems are an important part of an effective bias mitigation strategy, and to get that right, we'll need a lot of tools from our data science tool belt like adversarial models, tests for practical and statistical differences in group outcomes, tests for differential performance across demographic groups, and various bias remediation approaches. First, let's go over some terms that we'll be using throughout this chapter:

- **Bias:** For this chapter we mean *systemic biases* — historical, societal, and institutional, as defined by the National Institute of Standards and Technology (NIST) [SP 1270](#) artificial intelligence (AI) bias guidance.
- **Adversarial model:** In bias testing, we often train adversarial models on the predictions of the model we're testing to predict demographic information. If a ML model (the adversary) can predict demographic information from another model's predictions, then those predictions likely contain demographic information, and probably encode some amount of systemic bias. Crucially, the predictions of adversarial models also give us a row-by-row measure of bias. The rows for which the adversary model is most accurate likely encode more demographic information, or proxies thereof, than other rows.
- **Practical and statistical significance testing:** One of the oldest types of bias testing focuses on mean *outcome* differences across groups. We might use practical tests or effect size measurements, like adverse impact ratio (AIR) or standardized mean difference (SMD), to understand whether differences between mean outcomes are practically meaningful. We might use statistical significance testing to

understand whether mean differences across demographic groups are more associated with our current sample of data or are likely to be seen again in the future.

- **Differential performance testing:** Another common type of testing is to investigate performance differences across groups. We might investigate whether true positive rates (TPR), true negative rates (TNR), R^2 , or root mean squared error (RMSE) are roughly equal, or not, across demographic groups.
- **Four-fifths rule:** The four-fifth's rule is a guideline released in the 1978 [Uniform Guidelines on Employee Selection Procedures](#) (UGESP) by the Equal Employment Opportunity Commission (EEOC). Part 1607.4 of the UGESB states that, “[a] selection rate for any race, sex, or ethnic group which is less than four-fifths (4/5) (or eighty percent) of the rate for the group with the highest rate will generally be regarded by the Federal enforcement agencies as evidence of adverse impact.” For better or worse, the value of 0.8 for AIR — which compares event rates, like job selection or credit approval — has become a widespread benchmark for bias in ML systems.
- **Remediation approaches:** When testing identifies problems, we'll want to fix them. Technical bias mitigation approaches are often referred to as *remediation*. One thing we can say about ML models and bias is that ML models seem to present more ways to fix themselves than traditional linear models. Due to the *Rashomon effect* - the fact that there are often many accurate ML models for any given training dataset - we simply have more levers to pull and switches to flip to find better options for decreased bias and sustained predictive performance in ML models versus simpler models. Since there are so many options for models in ML, there are many potential ways to remediate bias. Some of the most common include pre-, in-, and post-processing, and model selection.
 - **Pre-processing:** Rebalancing, reweighing, or re-sampling training data so that demographic groups are better represented or positive outcomes are distributed more equitably.
 - **In-processing:** Any number of alterations to ML training algorithms, including constraints, regularization and dual loss functions, or incorporation of adversarial modeling information, that attempt to generate more balanced outputs or performance across demographic groups.
 - **Post-processing:** Changing model predictions directly to create less biased outcomes.
 - **Model selection:** Considering bias along with performance when selecting models. Typically it's possible to find a model with good performance and fairness characteristics if we measure bias and performance across a large set of hyperparameter settings and input features.

Finally, we'll need to remember that legal liability can come into play with ML bias issues. There are many legal liabilities associated with bias in ML systems, and since we're not lawyers (and likely neither are readers), we need to be humble about the complexity of law, not let the Dunning-Kruger effect take over, and defer to actual experts on nondiscrimination law. If we have any concerns about legal problems in our ML systems, now is the time to reach out to our managers or our legal department. With all this serious information in mind, let's now jump into training an XGBoost model, and testing it for bias.

Model Training

The first step in this chapter's use case is to train an XGBoost model on the credit card example data. To avoid disparate treatment concerns, we will not be using demographic features as inputs to this model. Generally speaking, for most business applications, it's safest not to use demographic information as model inputs. Not only is this legally risky in spaces like consumer credit, housing and employment, it also implies that business decisions should be based on race or gender — and that's dangerous territory. It's also true, however, that using demographic data in model training can decrease bias, and we'll see a version of that when we try out in-processing bias remediation below. There also may be certain kinds of decisions that should be based on demographic information, such as in medical treatments. Since this is an example credit decision, and since we're not sociologists or nondiscrimination law experts, we're going to play it safe and not use demographic features in our model. We will be using demographic features to test for bias and to remediate bias later in the chapter.

```
id_col = 'ID'  
groups = ['SEX', 'RACE', 'EDUCATION', 'MARRIAGE', 'AGE']  
target = 'DELINQ_NEXT'  
features = [col for col in train.columns if col not in groups + [id_col, target]]
```

Despite its risks, demographic information is important for bias management, and one way organizations go wrong in managing ML bias risks is by not having the necessary information on-hand to test and then remediate bias. At minimum, this means having people's names and zip codes, so that we could use [Bayesian improved surname geocoding](#), and related techniques, to infer their demographic information. If data privacy controls allow, and the right security is place, it's most useful for bias-testing to collect people's demographic characteristics directly. It's important to note that all the techniques used below do require demographic information, but for the most part, we can use demographic information that is inferred or collected directly. With these important caveats addressed, let's look at training our constrained XGBoost model and selecting a score cutoff.



One place where we as data scientists tend to go wrong is by using demographic information in models or technical bias remediation approaches in a way that could amount to *disparate treatment*. Adherents to the *fairness through awareness* doctrine may rightly disagree in some cases, but as of today, the most conservative approach to bias management in ML related to housing, credit, employment and other traditional high-risk applications is to use no demographic information directly in models or bias remediation. Using demographic information only for bias testing is generally acceptable. See [Chapter 4](#) for more information.



Before training a model in a context where bias risks must be managed, we should always make sure that we have the right data on hand to test for bias. At minimum, this means name, zip code, and a BISG implementation. At maximum, it means collecting demographic labels and all the data privacy and security that goes along with collecting and storing sensitive data. Either way, ignorance is not bliss when it comes to ML bias.

We will be taking advantage of monotonicity constraints again. A major reason transparency is important with respect to managing bias in ML is that if bias testing highlights issues — and it often does — we have a better chance of understanding what's broken about the model, and if we can fix it. If we're working with an unexplainable ML model, and bias problems emerge, we often end up scrapping the whole model and hoping for better luck in the next unexplainable model. That doesn't feel very scientific to us.

We like to test, debug, and understand, to the extent possible, how and why ML models work. In addition to being more stable and more generalizable, our constrained XGBoost model should also be more transparent and debuggable. We also have to highlight that when we take advantage of monotonicity constraints to enhance explainability *and* XGBoost's custom objective functionality to consider performance and bias simultaneously (see "[In-processing](#)" on page 290 bias remediation below), we're modifying our model to be both more transparent *and* more fair. Those seem like the exact right kind of changes to make if we're worried about stable performance, maximum transparency, and minimal bias in a high-risk application. It's great that XGBoost is mature enough to offer this level of deep customizability. (Unfortunately for readers working in credit, mortgage, housing, employment and other traditionally regulated sectors, you likely need to check with legal departments before employing a custom objective function that processes demographic data due risks of disparate treatment.)



We can combine monotonicity constraints (enhanced explainability) and customized objective functions (bias management) in XGBoost to directly train more transparent and less biased ML models.

In terms of defining the constraints for this chapter, we use a basic approach based on Spearman correlation. Spearman correlation is nice because it considers monotonicity rather than linearity (as is the case in the Pearson correlation coefficient). We also implement a `corr_threshold` argument to our constraint selection process so that small correlations don't cause spurious constraints.

```
def get_monotone_constraints(data, target, corr_threshold):

    # determine Spearman correlation
    # create a tuple of 1,0,-1 for each feature
    # 1 - positive constrain, 0 - no constraint, -1 - negative constraint
    corr = pd.Series(data.corr(method='spearman')[target]).drop(target)
    monotone_constraints = tuple(np.where(corr < -corr_threshold,
                                           -1,
                                           np.where(corr > corr_threshold, 1,
                                                   0)))
    return monotone_constraints

# define constraints
correlation_cutoff = 0.1
monotone_constraints = get_monotone_constraints(train[features+[target]],
                                                 target,
                                                 correlation_cutoff)
```

To train the model, our code is very straightforward. We'll start with hyperparameters we've used before to good result and not go crazy with hyperparameter tuning. We're just trying to start off with a decent baseline because we'll be doing a lot of model tuning and applying careful selection techniques when we get into bias remediation. Here's what our first attempt at training looks like.

```
# feed the model the global bias
# define training params, including monotone_constraints
base_score = train[target].mean()

params = {
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'eta': 0.05,
    'subsample': 0.6,
    'colsample_bytree': 1.0,
    'max_depth': 5,
    'base_score': base_score,
    'monotone_constraints': dict(zip(features, monotone_constraints)),
    'seed': seed
}
```

```
# train using early stopping on the validation dataset.  
watchlist = [(dtrain, 'train'), (dvalid, 'eval')]  
model_constrained = xgb.train(params,  
                               dtrain,  
                               num_boost_round=200,  
                               evals=watchlist,  
                               early_stopping_rounds=10,  
                               verbose_eval=False)
```

To calculate test values like AIR and other performance quality ratios across demographic groups in subsequent sections, we'll need to establish a probability cutoff so that we can measure our model's outcomes and not just its predicted probabilities. Much like when we train the model, we're looking for a starting point to get some baseline readings right now. We'll do that using common performance metrics like F1, precision, and recall. In [Figure 8-1](#) you can see that by picking a probability cutoff that maximizes F1, we make a solid tradeoff between precision — the model's proportion of *positive decisions* that are correct (positive predicted value) — and recall — the model's proportion of *positive outcomes* that are correct (true positive rate). For our model, that number is 0.26. To start off, all predictions above 0.26 are not going to get the credit line increase on offer. All predictions that are 0.26 or below will be accepted.

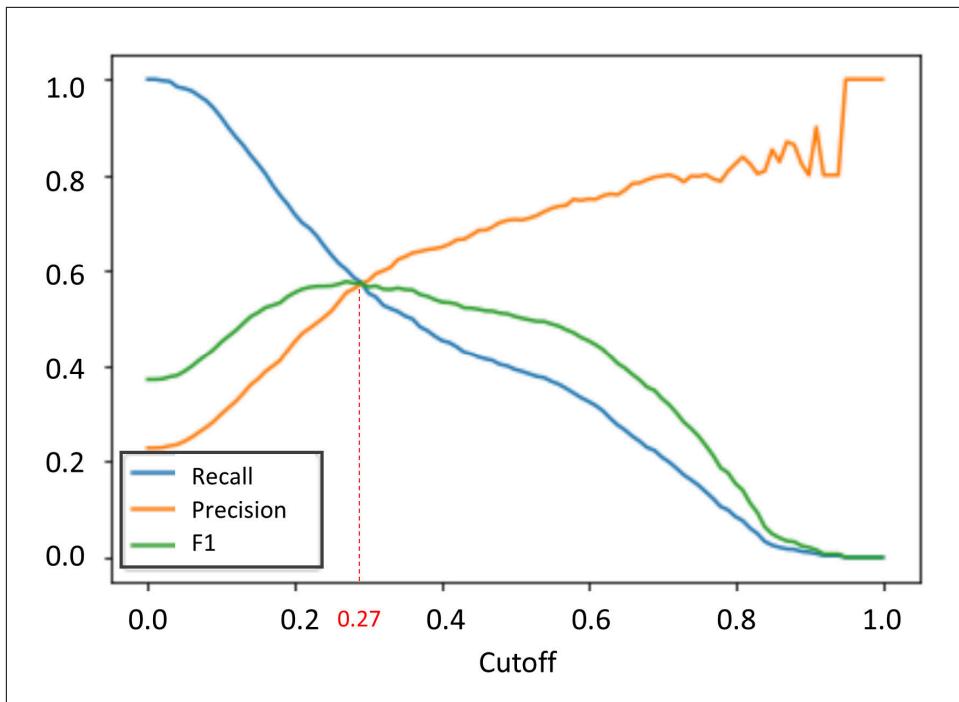


Figure 8-1. A preliminary cutoff, necessary for initial bias testing, is selected by maximizing F1 statistic.

We know that we'll likely end up tuning the cutoff due to bias concerns as well. In our data and example setup, increasing the cutoff means lending to more people. When we increase the cutoff, we hope that we are also lending to more different kinds of people. When we decrease the cutoff, we make our credit application process more selective, lending to fewer people, and likely fewer different kinds of people too. Another important note about cutoffs — if we're monitoring or auditing an already-deployed ML model, we should use the exact cutoff that is used for *in vivo* decision-making, not an idealized cutoff based on performance statistics like we've selected here.



In training and monitoring credit models, we have to remember that we typically only have good data for applicants that were selected in the past for a credit product. Most agree that this phenomenon introduces bias into any decision based only on previously selected individuals. What to do about it, widely discussed as *reject inference* techniques, is less clear. Keep in mind, similar bias issues apply to other types of applications where long-term data about unselected individuals is not available.

Evaluating Models for Bias

Now that we have a model and a cutoff, let's dig in and start to test it for bias. In this section, we'll test for different types of bias: bias in performance, bias in outcome decisions, bias against individuals, and proxy bias. First we construct confusion matrices and many different performance and error metrics for each demographic group. We'll apply established bias thresholds from employment as a rule of thumb to ratios of those metrics to identify any problematic bias in performance. We'll then apply traditional bias tests and effect size measures, aligned with those used in U.S. fair lending and employment compliance programs, to test model outcomes for bias. From there we'll look at residuals to identify any outlying individuals or any strange outcomes around our cutoff. We'll also use an adversarial model to identify any rows of data that seem to be encoding more bias than others. We'll close out our bias testing discussion by highlighting ways to find proxies, i.e., seemingly neutral input features that act like demographic information in models, that can lead to different types of bias problems.

Testing Approaches for Groups

We'll start off our bias-testing exercise by looking for problems in how our model treats groups of people on average. We'll be looking at model performance and whether it's roughly equivalent across groups or not. We'll also be testing for the absence of *group fairness*, sometimes also called *statistical or demographic parity*, in model outcomes. These notions of group fairness are flawed, because defining and measuring groups of people is difficult (or impossible). Furthermore, averages hide a lot of information about individuals, and the thresholds used for these tests are somewhat arbitrary. Despite these shortcomings, these tests are some of the most common used today. They can tell us useful information about how our model behaves at a high-level and point out serious areas of concern. Like many of the tests we'll discuss in this section, the key to interpreting them is as follows: *passing these tests doesn't mean much — our model or system could still have serious in vivo bias issues — but failing them is a big red flag for bias.*

Before jumping into the tests themselves, it's important to think about where to test. Should we test in training, validation or test data? The most standard partitions in which to test are validation and test data, just like we test our model's performance. Testing for bias in validation data can also be used for model selection purposes, as we'll discuss below. Using test data should give us some idea of how our model will perpetuate bias once deployed. (There are no guarantees an ML model will perform similarly to what we observe in test data, so monitoring for bias after deployment is crucially important.) Bias-testing in training data is mostly useful for observing differences in bias measurements from validation and test partitions. This is especially helpful if one partition stands out from the others, and can potentially be used for

understanding drivers of bias in our model. If training, validation, and test sets are constructed so that train comes first in time and test comes last — as they likely should be — comparing bias measurements across data partitions is also helpful for understanding trends in bias. It is a concerning sign to see bias measurements increase from train to validation to test. One other option is to estimate variance in bias measurements using cross validation or boot strapping, as is done with standard performance metrics too. Cross-validation, boot strapping, standard deviations or errors, confidence intervals, and other measure of variance for bias metrics can help us understand if our bias-testing results are more precise or more noisy — an important part of any data analysis.



We must be absolutely clear about how a positive decision is represented in the data, what positive means in the real-world, how our model's predicted probabilities align to these two notions — and which cutoffs generate positive decisions — before we can begin bias testing. In our example, the decision that is desirable to the preponderance of model subjects is an outcome of zero, associated with probabilities below the cutoff value of 0.26. Observations who receive a classification of zero will be extended a line of credit.

In the bias testing conducted below, we'll be sticking to basic practices, and looking for biases in model performance and outcomes in validation and test data. If you've never tried bias testing, this is a good way to get started. And inside large organizations, where logistics and politics make bias testing even more difficult, this might be the only bias testing that can be accomplished. Bias testing is also never finished. As long as model is deployed, it needs to be monitored and tested for bias. All of these practical concerns make bias testing a big effort, and for these reasons we'd urge you to begin with these standard practices that look for bias in performance and outcomes across large demographic groups, and then use any remaining time, resources, or will to investigate bias against individuals and to identify proxies or other drivers of bias in your model. That's what we'll get into now.

Testing Performance

A model should have roughly similar performance across demographic groups, and if doesn't, this is an important type of bias. If all groups are being held to the same standard by an ML model for receiving a credit product, but that standard is not an accurate predictor of future repayment behavior for some groups, that's not fair. (This is somewhat similar to the employment notion of *differential validity*, discussed in [Chapter 4](#).) To start testing for bias in performance across groups for a binary classifier like our XGBoost model, we'll look at confusion matrices for each group and form different measures of performance and error across groups. We'll consider com-

mon measures like true positive and false positive rates, as well as some less common in data science, like false discovery rate.

The code block below is far from the best implementation, because of its reliance on dynamic code generation and an `eval()` statement, but it is written to be maximally illustrative. In it readers can see how the four cells in a confusion matrix can be used to calculate many different performance and error metrics.

```
def confusion_matrix_parser(expression):

    # tp / fp      cm_dict[level].iat[0, 0] / cm_dict[level].iat[0, 1]
    # ----- ==> -----
    # fn / tn      cm_dict[level].iat[1, 0] / cm_dict[level].iat[1, 1]

    metric_dict = {
        'Prevalence': '(tp + fn) / (tp + tn + fp + fn)',
        'Accuracy': '(tp + tn) / (tp + tn + fp + fn)',
        'True Positive Rate': 'tp / (tp + fn)',
        'Precision': 'tp / (tp + fp)',
        'Specificity': 'tn / (tn + fp)',
        'Negative Predicted Value': 'tn / (tn + fn)',
        'False Positive Rate': 'fp / (tn + fp)',
        'False Discovery Rate': 'fp / (tp + fp)',
        'False Negative Rate': 'fn / (tp + fn)',
        'False Omissions Rate': 'fn / (tn + fn)'
    }

    expression = expression.replace('tp', 'cm_dict[level].iat[0, 0]')\
        .replace('fp', 'cm_dict[level].iat[0, 1]')\
        .replace('fn', 'cm_dict[level].iat[1, 0]')\
        .replace('tn', 'cm_dict[level].iat[1, 1]')

    return expression
```

When we apply the `confusion_matrix_parser` function to confusion matrices for each demographic group along with other code that loops through groups and the measures in `metric_dict`, we can make a table like [Table 8-1](#) below. For brevity, we've focused on the race measurements in this subsection. If this were a real credit or mortgage model, we'd be looking at different genders, different age groups, those with disabilities, different geographies and maybe even other subpopulations.

Table 8-1. Common performance and error measures derived from a confusion matrix across different race groups for test data.

Group	Prevalence	Accuracy	True Positive Rate	Precision	Specificity	Negative Predicted Value	False Positive Rate	False Discovery Rate	False Negative Rate	False Omissions Rate
Hispanic	0.399393	0.725986	0.637975	0.663158	0.784512	0.765189	0.215488	0.336842	0.362025	0.234811
Black	0.386707	0.720040	0.635417	0.638743	0.773399	0.770867	0.226601	0.361257	0.364583	0.229133

Group	Prevalence	Accuracy	True Positive Rate	Precision	Specificity	Negative Predicted Value	False Positive Rate	False Discovery Rate	False Negative Rate	False Omissions Rate
White	0.107075	0.829828	0.470238	0.307393	0.872948	0.932165	0.127052	0.692607	0.529762	0.067835
Asian	0.101010	0.853199	0.533333	0.350877	0.889139	0.944312	0.110861	0.649123	0.466667	0.055688

Table 8-1 starts to show us some hints of bias in our model's performance, but it's not really measuring bias yet. It's simply showing the value for different measurements across groups. We should start to pay attention when these values are obviously different for different groups. For example, precision looks quite different between demographic groups (White and Asian people on one hand, and Black and Hispanic people on the other). The same can be said about other measures like false positive rate, false discovery rate, and false omissions rate. (Disparities in prevalence tell us that default occurs more *in the data* for Black and Hispanic people. Sadly this not uncommon in many U.S. credit markets.) In **Table 8-1**, we are starting to get a hint that our model is predicting more defaults for Black and Hispanic people, but it's still hard to tell if it's doing a good or equitable job. (Just because a dataset records these kinds of values, does not make them objective or fair!) To help understand if the patterns we're seeing are actually problematic we need to take one more step. We'll follow methods from traditional bias testing and divide the value for each group by the corresponding value for the control group and apply the four-fifths rule as a guide. In this case, we *assume* the control group is White people.



Strictly speaking, in the employment context, the control group is the most favored group in an analysis, not necessarily White people or males. There may also be other reasons to use control groups that are not White people or males. Choosing the control or reference group for a bias testing analysis is a difficult tasks, best done in concert with legal, compliance, social science experts, or stakeholders.

Once we do this division, we see the values in **Table 8-2**. (We divide each column in the table by the value in the White row. That's why the White values are all 1.0.) Now we can look for values outside of a certain range. We'll use the four-fifths rule, which has no legal or regulatory standing when used this way, to help us identify one such range: 0.8 — 1.25, or a 20% difference between groups. (Some prefer a tighter range of acceptable values, especially in high-risk scenarios, say 0.9 — 1.11, indicating a 10% difference between groups.) When we see values above 1 for these disparity measures, it means the protected or minority group has a higher value of the original measure, and vice versa for values below 1.

Looking at **Table 8-2**, we see no out-of-range values for Asian people. This means that the model performs fairly equitably across White and Asian people. However, we do

see glaring out-of-range values for Hispanic and Black people for precision, false positive rate, false discovery rate, and false omissions rate disparity. While applying the four-fifths rule can help us flag these values, it really can't help us interpret them. For this, we'll have to rely on our human brains to think through these results. We also need to remember that a decision of 1 from our model is a predicted default, and that higher probabilities mean default is more likely in the eyes of the model.

Table 8-2. Performance-based bias measures across race groups for test data.

Group	Prevalence Disparity	Accuracy Disparity	True Positive Rate Disparity	Precision Disparity	Specificity Disparity	Negative Predicted Value Disparity	False Positive Rate Disparity	False Discovery Rate Disparity	False Negative Rate Disparity	False Omissions Rate Disparity
Hispanic	3.730048	0.874863	1.356706	2.157362	0.898693	0.820873	1.696062	0.486339	0.683374	3.461486
Black	3.611567	0.867698	1.351266	2.077938	0.885962	0.826965	1.783528	0.521590	0.688202	3.377775
White	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
Asian	0.943362	1.028163	1.134177	1.141461	1.018547	1.013031	0.872567	0.937217	0.880899	0.820931

Given that prevalence of defaults in the data is so much higher for Black and Hispanic people, one thing these results suggest is that our model learned more about defaults in these groups, and predicts defaults at a higher rate in these groups. Traditional testing in the next section will try to get at the underlying question of whether it's fair to predict more defaults in these groups. For now, we're trying to figure out if the performance of the model is fair. Looking at which measures are out-of-range for protected groups and what they mean, we can say:

- Precision disparity: ~2X (more) correct default predictions, out of those *predicted to default*.
- False positive rate disparity: ~1.5X (more) incorrect default predictions, out of those *that did not default*.
- False discovery rate disparity: ~0.5X (less) incorrect default predictions, out of those *predicted to default*
- False omissions rate disparity: ~3.5X (more) incorrect acceptance predictions, out of those *predicted not to default*.

Precision and false discovery rate have the same denominator — the smaller group of those predicted to default — and can be interpreted together. They show this model has a higher rate of true positives for Black and Hispanic people relative to White people — meaning a higher rate of correct default predictions for this group. The false discovery rate echoes this result, pointing to a lower rate of false positives, or incorrect default decisions, for the minority groups in question. Relatedly, the false omissions rate shows our model makes incorrect acceptance decisions at a higher

rate, out of the larger group comprised of those predicted not to default, for Black and Hispanic people. Precision, false discovery rate, and false omissions rate disparity show serious bias issues, but a bias that favors Black and Hispanic people in terms of model performance.

Thinking Through the Confusion Matrix

In our use case, we put our brains to together to interpret the meaning of confusion matrix measures for our specific use case and context:

- **Prevalence:** how much default actually happens for this group.
- **Accuracy:** how often the model predicts default and non-default correctly for this group.
- **True Positive Rate:** out of the people in the group **that did** default, how many the model predicted **correctly** would default.
- **Precision:** out of the people in the group the model **predicted** would default, how many the model predicted **correctly** would default.
- **Specificity:** out of the people in the group **that did not** default, how many the model predicted **correctly** would not default.
- **Negative Predicted Value:** out of the people in the group the model **predicted** would not default, how many the model predicted **correctly** would not default.
- **False Positive Rate:** out of the people in the group **that did not** default, how many the model predicted **incorrectly** would default.
- **False Discovery Rate:** out of the people in the group the model **predicted** would default, how many the model predicted **incorrectly** would default.
- **False Negative Rate:** out of the people in the group **that did** default, how many the model predicted **incorrectly** would not default.
- **False Omissions Rate:** out of the people in the group the model **predicted** would not default, how many the model predicted **incorrectly** would not default.

Try to follow this example to create interpretations for confusion matrix performance and error measurements in your next important ML project. It can help to think through bias, performance, and safety issues with more clarity.

False positive rate disparity shows something a little different. False positive rate is measured out of the larger group of those who did not default, in reality. In that group, we do see higher rates of incorrect default decisions, or false positives, for Black and Hispanic people. Taken together, all these results point to a model with bias problems, some of which genuinely appear to favor minority groups. Of these, the false positive disparity is most concerning. It shows us that out of the relatively large

group of people who did not default, Black and Hispanic people are predicted to default incorrectly at 1.5X the rate of White people. This means a lot of historically disenfranchised people are being wrongly denied credit line increases by this model, which can lead to real-world harm. Of course, we also see evidence of correct and incorrect acceptance decisions favoring minorities. None of this is a great sign, but we need to dig into outcomes testing in the next section to get a clearer picture of group fairness in this model.



For regression models, we can skip the confusion matrices and proceed directly to comparing measures like R^2 or root mean squared error (RMSE) across groups. Where appropriate, and especially for measures bounded like R^2 or mean average percentage error (MAPE), we can also apply the four-fifth's rule as a rule of thumb to a ratio of these measures to help spot problematic performance bias.

In general, performance testing is a helpful tool for learning about wrong and negative decisions, like false positives. More traditional bias testing that focuses on outcomes rather than performance has a more difficult time highlighting bias problems in wrong or negative decisions. Unfortunately, as we're about to see, performance and outcomes testing can show different results. While some of these performance tests show a model that favors minorities, we'll see in the next section that's not true. Rates standardize out the raw numbers of people and raw scores of the model in theoretically useful ways. A lot of the positive results we saw here are for fairly small groups of people. When we consider real-world outcomes, the picture of bias in our model is going to be different and more clear. These kinds of conflicts between performance testing and outcomes testing are common and well documented, and we'd argue that outcomes testing — aligned with legal standards and what happens in the real-world — is more important.



There is a well-known tension between improved performance in data that encode historical biases — like most of the data we work with — and balancing outcomes across demographic groups. Remember, data is often *not* objective. If we make outcomes more balanced, this tends to decrease performance metrics in a biased dataset.

Because it's difficult to interpret all these different performance measures, some may have more meaning in certain scenarios than others, and they are likely to be in conflict with each other or outcomes testing results, prominent researchers put together a **decision tree** (slide 40) to help focus on a smaller subset of performance disparity measures. According to this tree, where our model is punitive (higher probability means default/reject decision), and the clearest harm is incorrectly denying credit line

increases to minorities (intervention not warranted), the false positive rate disparity should probably carry the highest weight in our prediction performance analysis. The false positive rate disparity doesn't tell a nice story. Let's see what outcomes testing shows.

Traditional Testing of Outcomes Rates

The way we set up our analysis, based on a binary classification model, it was easiest to look at *performance* across groups first using confusion matrices. What's likely more important, and likely more aligned to legal standards in the U.S., is to analyze differences in *outcomes* across groups using traditional measures of statistical and practical significance. We'll pair two well-known practical bias-testing measures — AIR and SMD, with chi-squared and *t*-tests, respectively. Understanding whether a discovered difference in group outcomes is statistically significant is usually a good idea, but in this case, it might also be a legal requirement. Statistically significant differences in outcomes or mean scores is one of the most common legally recognized measures of discrimination, especially in areas like credit lending where algorithmic decision-making has been regulated for decades. By using practical tests and effect size measures, like AIR and SMD, with statistical significance tests we get two pieces of information: the magnitude of the observed difference, and whether it's statistically significant, i.e., likely to be seen again in other samples of data.



If you're working in a regulated vertical or in a high-risk application, it's a good idea to apply traditional bias tests with legal precedent first before applying newer bias-testing approaches. Legal risks are often the most serious organizational risks for many types of ML-based products and laws are often designed to protect users and stakeholders.

AIR is often applied to categorical outcomes, like credit lending or hiring outcomes, where someone either receives a positive outcome or not. AIR is defined as the rate of positive outcomes for a protected group, like minorities or women, divided by the same rate of positive outcomes for a control group, like White people or men. According to the four-fifths rule, we look for the AIR to be above 0.8. An AIR below 0.8 points to a serious problem. We then test whether this difference will probably be seen again or if it's due to chance using a chi-squared test.



Impact ratios can also be used for regression models by dividing average scores for a protected group by average scores for a control group, and applying the four-fifths rule as a guideline for identifying problematic results. Other traditional bias measurement approaches for regression models in *t*-tests and SMD.

While AIR and chi-squared are most often used with binary classification, SMD and t-tests are often used on predictions from regression models, or on numeric quantities like wages, salaries, or credit limits. We'll apply SMD and *t*-tests to our model's predicted probabilities for demonstration purposes and to get some extra information about bias in our model. SMD is defined as the mean score for a protected group minus the mean score for a control group, with that quantity divided by a measure of the standard deviation of the score. SMD has well-known cutoffs at a magnitudes of 0.2, 0.5, and 0.8 for small, medium, and large differences, respectively. We'll use a *t*-test to decide whether the effect size measured by SMD is statistically significant.



This application of SMD - applied to the probabilities output by the model - would also be appropriate if the model scores would be fed into some downstream decision-making process, and it is impossible to generate model outcomes at the time of bias testing.

In addition to significance tests, AIR, and SMD, we'll also be analyzing basic descriptive statistics like counts, means and standard deviations, as can be seen in [Table 8-3](#). When looking over [Table 8-3](#), it's clear that there is a big difference in scores for Black and Hispanic people versus scores for White and Asian people. While our data is simulated, very sadly, this is not atypical in U.S. consumer finance. Systemic bias is real, and fair lending data tends to prove it. (If you'd like to satisfy your own curiosity on this matter, we urge you to analyze some [freely available](#) Home Mortgage Disclosure Act (HMDA) data.)

Table 8-3. Traditional outcomes-based bias metrics across race groups for test data.

Group	Count	Favorable Outcomes	Favorable Rate	Mean Score	Std. Dev. Score	AIR Score	AIR p-value	SMD	SMD p-value
Hispanic	989	609	0.615774	0.291994	0.205374	0.736394	6.803832e-36	0.528829	4.311292e-35
Black	993	611	0.615307	0.279935	0.199937	0.735836	4.343885e-36	0.482898	4.564909e-30
Asian	1485	1257	0.846465	0.177847	0.169855	0.012274	4.677908e-01	-0.032609	8.162671e-01
White	1569	1312	0.836201	0.183386	0.172153	1.000000	-	0.000000	-

In [Table 8-3](#), it's immediately obvious that Black and Hispanic people have higher mean scores and lower favorable rates than White and Asian people, while all four groups have similar standard deviations for scores. Are these differences big enough to be a bias problem? That's where our practical significance tests come in. AIR and SMD are both calculated in reference to White people. That's why White people have scores of 1.0 and 0.0 for these, respectively. Looking at AIR, both Black and Hispanic AIRs are below 0.8. Big red flag! SMD for those two groups are around 0.5, meaning a medium difference in scores between groups. That's not a great sign either. We'd like for those SMD values to be below or around 0.2, signifying a small difference.



AIR is often misinterpreted by data scientists. Here's a simple way to think of it: An AIR value above 0.8 doesn't mean much, and it certainly doesn't mean a model is fair. However, AIR values below 0.8 point to a serious problem.

The next question we might ask in a traditional bias analysis is whether these practical differences for Black and Hispanic people are statistically significant. Bad news — they are very significant, with p -values approaching 0 in both cases. While datasets have exploded in size since the 1970s, a lot of legal precedent points to statistical significance at the 5% level ($p = 0.05$) for a two-sided hypothesis test as a marker of legally impermissible bias. Since this threshold is completely impractical for today's large datasets, we recommend adjusting p -value cutoffs lower for larger datasets. However, we should also be prepared to be judged at $p = 0.05$ in regulated verticals of the U.S. economy. Of course, fair lending and employment discrimination cases are anything but straightforward, and facts, context, and expert witnesses have as much to do with a final legal determination as any bias-testing number. An important takeaway here is that the law in this area is already established, and not as easily swayed by artificial intelligence (AI) hype as internet and media discussions. If we're operating in a high-risk space, we should probably conduct traditional bias tests in addition to newer tests, as we've done here.



In consumer finance, housing, employment and other traditionally regulated verticals of the U.S. economy, nondiscrimination law is highly mature and not swayed by AI hype. Just because AIR and two-sided statistical tests feel outdated or simplistic to data scientists, does not mean our organizations won't be judged by these standards if legal issues arise.

These race results point to a fairly serious discrimination problem in our model. If we were to deploy it, we'd be setting ourselves up for potential regulatory and legal problems. Worse than that, we'd be deploying a model we know perpetuates systemic biases and harms people. Getting an extension on a credit card can be a serious thing at different junctures in our lives. If someone is asking for credit, we should assume it's genuinely needed. What we see here is that an example credit-lending decision is tinged with historical biases. These results also send a clear message. This model needs to be fixed before it's deployed.

Individual Fairness

We've been focused on group fairness thus far, but we should also probe our model for individual fairness concerns. Unlike bias against groups, individual bias is a local issue that affects only a small and specific group people, down to a single individual. There are two main techniques we'll use to test this: residual analysis and adversarial

modeling. In the first technique — residual analysis — we look at individuals very close to the decision cutoff and who incorrectly received unfavorable outcomes as a result. We want to make sure their demographic information isn't pushing them into being denied for a credit product. (We can sanity check very wrong individual outcomes faraway from the decision cutoff too.) In the second approach — adversarial models — we'll use separate models that try to predict protected group information using the input features and the scores from our original model, and we'll look at those model's Shapley additive explanations (SHAP). When we find rows where adversarial predictions are very accurate, this is a hint that something in that row is encoding information that leads to bias in our original model. If we can identify what that something is across more than a few rows of data, we're on the path to identifying potential drivers of proxy bias in our model. We'll look into individual bias and then proxy bias before transitioning to the bias remediation section of the chapter.

Let's dive into individual fairness. First, we wrote some code to pull out a few narrowly misclassified people from a protected group. These are observations that our model predicted would go delinquent, but they did not.

```
black_obs = valid.loc[valid['RACE'] == 'black'].copy()
black_obs[f'p_{target}_outcome'] = np.where(black_obs[f'p_{target}'] >
best_cut, 1, 0)

misclassified_obs = black_obs[(black_obs[target] == 0) &
                             (black_obs[f'p_{target}_outcome'] == 1)]

misclassified_obs.sort_values(by=f'p_{target}').head(3)[features]
```

The results are shown in [Table 8-4](#), and they don't suggest any egregious bias, but they do raise some questions. The first and third applicants appear to spending moderately and making payments on time for the most part. These individuals may have been placed on the wrong side of a decision boundary in an arbitrary manner. However, the individual in the second row of [Table 8-4](#) appears not to be making progress on reaping their credit card debt. Perhaps they really should not have been approved for an increased line of credit.

Table 8-4. A subset of features for narrowly misclassified protected observations in validation data.

LIMIT_BAL	PAY_0	PAY_2	PAY_3	...	BILL_AMT1	BILL_AMT2	BILL_AMT3	...	PAY_AMT1	PAY_AMT2	PAY_AMT3
\$58,000	-1	-1	-2	...	\$600	\$700	\$0	...	\$200	\$700	\$0
\$58,000	0	0	0	...	\$8,500	\$5,000	\$0	...	\$750	\$150	\$30
\$160,000	1	-1	-1	...	\$0	\$0	\$600	...	\$0	\$0	\$0

Next steps to uncovering whether we've found a real individual bias problem might include:

- **Small perturbations of input features:** If some arbitrary change to an input feature, say decreasing BILL_AMT1 by \$5, changes the outcome for this person, then the model's decision may be more related to a steep place in its response function intersecting with the decision cutoff than any tangible real-world reason.
- **Searching for similar individuals:** If there are a handful — or more — individuals like the current individual, the model maybe segmenting some specific or intersectional subpopulation in an unfair or harmful way.

If either of these are the case, the right thing to do may be to extend this and similar individual's line(s) of credit.

We conducted a similar analysis for Hispanic and Asian observations and found similar results. We weren't too surprised by these results, for at least two reasons. First, individual fairness questions are difficult and bring up issues of causality which ML systems tend not to address in general. Second, individual fairness (and proxy discrimination) are probably much larger risks for datasets with many rows — where entire subpopulations may end up on an arbitrary side of a decision boundary — and when a model contains many features, and especially *alternative data*, or features not directly linked to one's ability to repay credit that may otherwise enhance the predictiveness of the model.



Answering questions about individual fairness with 100% certainty is difficult, because they're fundamentally *causal* questions. For complex, nonlinear ML models, it's impossible to know whether a model made a decision on the basis of some piece of data (i.e., protected group information) that isn't included in the model in the first place.

That said, residual analysis, adversarial modeling, SHAP values and the careful application of subject matter expertise can go a long way. For more reading on this subject, check out [Explaining quantitative measures of fairness](#) from the creator of SHAP values, and [On Testing for Discrimination Using Causal Models](#).

Let's move onto the second technique for testing individual fairness: adversarial modeling. We chose to train two adversarial models. The first model takes in the same input features as the original model, but attempts to predict protected groups status rather than delinquencies. For simplicity, we trained a binary classifier on a target for protected class membership — a new marker for Black or Hispanic people. By analyzing this first adversarial model, we can get a good idea of which features have the strongest relationships with protected demographic group membership.

The second adversarial model we train is exactly like the first, except it gets one additional input feature — the output probabilities of our original lending model. By

comparing the two adversarial models, we will get an idea of how much *additional* information was encoded in the original model scores. And we'll get this information at the observation level.



Many ML tools that generate *row-by-row* debugging information — like residuals, adversarial model predictions or SHAP values — can be used for examining individual bias issues.

We trained these adversarial models as binary XGBoost classifiers with similar hyperparameters to the original model. First, we took a look at the protected observations whose adversarial model scores increased the most when the original model probabilities are added as a feature. The results are shown in [Table 8-5](#). This table is telling us that for some observations, the original model scores are encoding enough information about protected group status that the second adversarial model is able to improve on the first by around 30 percentage points. These results tell us that we should take a deeper look into these observations, in order to identify any individual fairness problems by asking questions like we did for individual bias issues spotted with residuals. [Table 8-5](#) also helps us show again that removing *demographic markers* from a model does not remove *demographic information* from a model.

Table 8-5. The three protected observations that saw their scores increase the most between the two adversarial models in validation data.

Observation	Protected	Adversary 1 Score	Adversary 2 Score	Difference
9022	1	0.288	0.591	0.303
7319	1	0.383	0.658	0.275
528	1	0.502	0.772	0.270

Recall from [Chapter 2](#) that SHAP values are a row-by-row additive feature attribution scheme. That is, they tell us how much each feature in a model contributed to the overall model prediction. We computed the SHAP values on validation data for the second adversarial model (the one that includes our original model scores). In [Figure 8-2](#), we took a look at the distribution of SHAP values for the top four most important features. Each of the features in [Figure 8-2](#) is important to predicting protected class membership. Coming in as the most important feature for predicting protected group information is the original model scores, `p_DELINQ_NEXT`. This is interesting in and of itself, and the observations that have the highest shap values for this feature are good targets to investigate further for individual fairness violations.

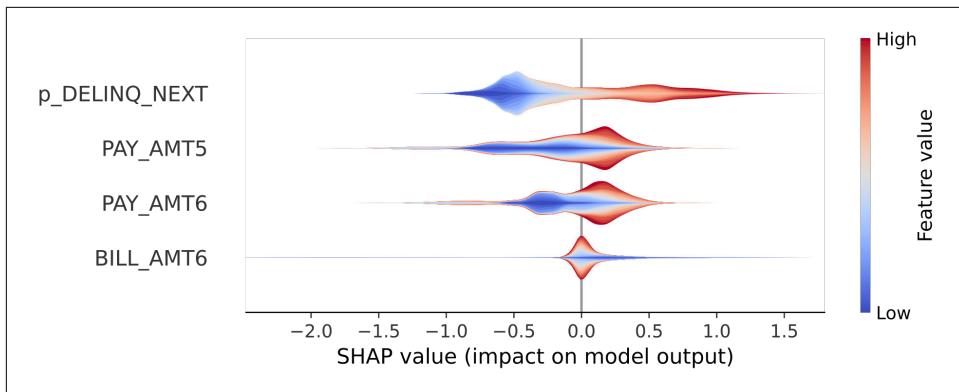


Figure 8-2. The distribution of SHAP values for the five most important features in our adversarial model in validation data.

The distribution of SHAP values for the five most important features in our adversarial model. Maybe more interesting is the color gradient (change from light to dark) within the `p_DELINQ_NEXT` violin. Each violin is colored by the value of the feature itself for each observation in the density. That means that if our model was linear with no interactions, the color gradient across each violin would be smooth from light to dark. But that's not what we observed. Within the `p_DELINQ_NEXT` violin, there is significant color variation within vertical slices of the plot. This can only arise when `p_DELINQ_NEXT` is being used by the model in conjunction with other features in order to drive the predictions. For example, the model might be learning something like *if LIMIT_BAL is below \$20,000 and if credit utilization is above 50% and if the delinquency probability from the credit extension model is above 20% then the observation is likely to be a Black or Hispanic person*. While residuals and adversarial models can help us identify individual bias issues, SHAP can take us a step further by helping us understand what is driving that bias.

Proxy Bias

If the patterns like those identified above only affect a few people, they can still be harmful. But when we see them affecting larger groups of people, we likely have a more global proxy bias issue on our hands. Remember that proxy bias happens when a single feature or a group of interacting features act like demographic information in our model. Given that ML models can often mix and match features to create latent concepts — and can do so in different ways on local, row-by-row basis (see above) — proxy bias is a fairly common driver of biased model outputs.

Many of the tools we discussed above like adversarial models and SHAP can be used to hunt down proxies. We could begin to get at them by looking at, for example, SHAP feature interaction values. (Recall advanced SHAP techniques from [Chapter 2](#)

and [Chapter 6](#).) The bottom line test for proxies may be adversarial models. If another model can accurately predict demographic information from our model's predictions, then our model contains demographic information. If we include model input features in our adversarial models, we can use feature attribution measures to understand which single input features might be proxies, and apply other techniques and elbow grease to find proxies created by interactions. Good, old-fashioned decision trees can be some of the best adversarial models for finding proxies. Since ML models tend to combine and recombine features, plotting a trained adversarial decision tree may help us uncover more complex proxies.

As readers can see, adversarial modeling can be a rabbit hole. But we hope we've convinced readers that it is a powerful tool for identifying individual rows that might be subject to discrimination under our models, and for understanding how our input features relate to protected group information and proxies. Now we're going to move onto the important job of remediating the bias we found in our example lending model.

Remediating Bias

Now that we've identified several types of bias in our model, it's time to roll up our sleeves and try to remediate it. Luckily there are many tools to choose from and, due to the dubious Rashomon Effect, many different models to choose from too. We'll try pre-processing remediation first. We'll generate observation-level weights for our training data so that positive outcomes appear equally likely across demographic groups. We'll then try an in-processing technique, sometimes known as *fair XGBoost*, in which demographic information is included in XGBoost's gradient calculation so that it can be regularized during model training. For post-processing, we'll update our predictions around the decision boundary of the model. Since pre-, in-, and post-processing may give rise to concerns about disparate treatment in several industry verticals and applications, we'll close out the remediation section by outlining a simple and effective technique for model selection that searches over various input feature sets and hyperparameter settings to find a model with good performance and minimal bias. For each approach, we'll also address any observed performance quality and bias remediation trade-offs.

Pre-processing

The first bias remediation technique we'll try is a preprocessing technique known as *reweighing*. It was published first by Faisal Kamiran and Toon Calders in their 2012 paper, [*Data Preprocessing Techniques for Classification Without Discrimination*](#). The idea of reweighing is to make the average outcome across groups equal using observation weights and then retrain the model. As we'll see below, before we pre-processed the training data, the average outcome, or average y variable value, was

quite different across demographic groups. The biggest difference was for Asian and Black people, with average outcomes of 0.107 and 0.400, respectively. This means that on average and looking only at the training data, Asian people's probability of default was well within the range or being accepted for a credit line increase, while the opposite was true for Black people. Their average score was solidly in the decline range. (Again, these values are not always objective or fair simply because they are recorded in digital data.) After we pre-process, we'll see we can balance out both outcomes and bias testing values to a notable degree.

Since reweighting is a very straightforward approach, we decided to implement it ourselves with the function in the code snippet below. (For an additional implementation and example usage of reweighing, checkout AIF360's [Detecting and mitigating age bias on credit decisions](#).) To reweigh our data, we first need to measure average outcome rates — overall and for each demographic group. Then we determine observation-level, or row-level, weights that balance out the outcome rate across demographic groups. Observation weights are numeric values that tell XGBoost, and most other ML models, how much to weigh each row during training. If a row has a weight of 2, it's like that row appears twice in the objective function used to train XGBoost. If we tell XGBoost that a row has a weight of 0.2, it's like that row appears 1/5 of the times it actually does in the training data. Given the average outcome for each group and their frequency in the training data, it's a basic algebra problem to determine the row weights that give all groups the same average outcome in the model.

```
def reweight_dataset(dataset, target_name, demo_name, groups):  
  
    n = len(dataset)  
  
    # initial overall outcome frequency  
    freq_dict = {'pos': len(dataset.loc[dataset[target_name] == 1])/n,  
                'neg': len(dataset.loc[dataset[target_name] == 0])/n}  
  
    # initial outcome frequency per demographic group  
    freq_dict.update({group: dataset[demo_name].value_counts()[group]/n for  
                      group in groups})  
  
    weights = pd.Series(np.ones(n), index=dataset.index)  
  
    # determine row weights that balance outcome frequency  
    # across demographic groups  
    for label in [0, 1]:  
        for group in groups:  
            label_name = 'pos' if label == 1 else 'neg'  
            freq = dataset.loc[dataset[target_name] == label]  
            [demo_name].value_counts()[group]/n  
            weights[(dataset[target_name] == label) & (dataset[demo_name] ==  
                  group)] *=\\  
                  freq_dict[group]*freq_dict[label_name]/freq
```

```
# return balanced weight vector
return weights
```

Applying the `reweight_dataset` function provides us with a vector of observation weights of the same length as the training data, such that a weighted average of the outcomes in the data within each demographic group is equal. Reweighting helps to undo manifestations of systemic biases in training data, teaching XGBoost that different kinds of people should have the same average outcome rates. In code, this is as simple as retraining XGBoost with the row weights from `reweight_dataset`. In our code, we call this vector of training weights `train_weights`. When we call the `DMatix` function, we use the `weight=` argument to specify these bias-decreasing weights. After this, we simply retrain XGBoost.

```
dtrain = xgb.DMatrix(train[features],
                      label=train[target],
                      weight=train_weights)
```

Table 8-6 below shows both the original mean outcomes and original AIR values, along with the preprocessed mean outcomes and AIR. When we trained XGBoost on the unweighted data, we see some problematic AIR values. Originally, the AIR for Hispanic people was around 0.73 for Black and Hispanic people. These values are not great — signifying that for every 1,000 credit products the model extends to White people, this model only accepts applications from about 730 Hispanic or Black people. This level of bias is ethically troubling, but it could also give rise to legal troubles in consumer finance, hiring, or other areas that rely on traditional legal standards for bias testing. The 4/5ths rule — while flawed and imperfect — tells us we should not see values below 0.8 for AIR. Luckily, in our case, reweighing provides good remediation results. Let's have a look at **Table 8-6**.

Table 8-6. Original and preprocessed mean outcomes for demographic groups in test data.

Demographic Group	Original Mean Outcome	Pre-processed Mean Outcome	Original AIR	Preprocessed AIR
Hispanic	0.398	0.22	0.736	0.861
Black	0.400	0.22	0.736	0.877
White	0.112	0.22	1.011	1.000
Asian	0.107	0.22	1.012	1.010

We increased the problematic AIR values for Hispanic and Black people to less borderline values, and importantly, without changing AIR very much for Asian people. In short, reweighing decreased potential bias risks for Black and Hispanic people, without increasing those risks for other groups. Did this have any effect on the performance quality of our model? To investigate this, we introduced a hyperparameter, `lambda` in **Figure 8-3** that dictates the strength of the reweighing scheme. When `lambda` is equal to zero, all observations get a sample weight of one. When the hyper-

parameter is equal to one, the mean outcomes are all equal and we get the results in [Table 8-6](#). As shown in [Figure 8-3](#), we did observe some trade-off between increasing the strength of reweighing and performance as measured by F1 in validation data. Next, let's look at the result on Black and Hispanic AIRs as we sweep `lambda` across a range of values to understand more about that tradeoff.

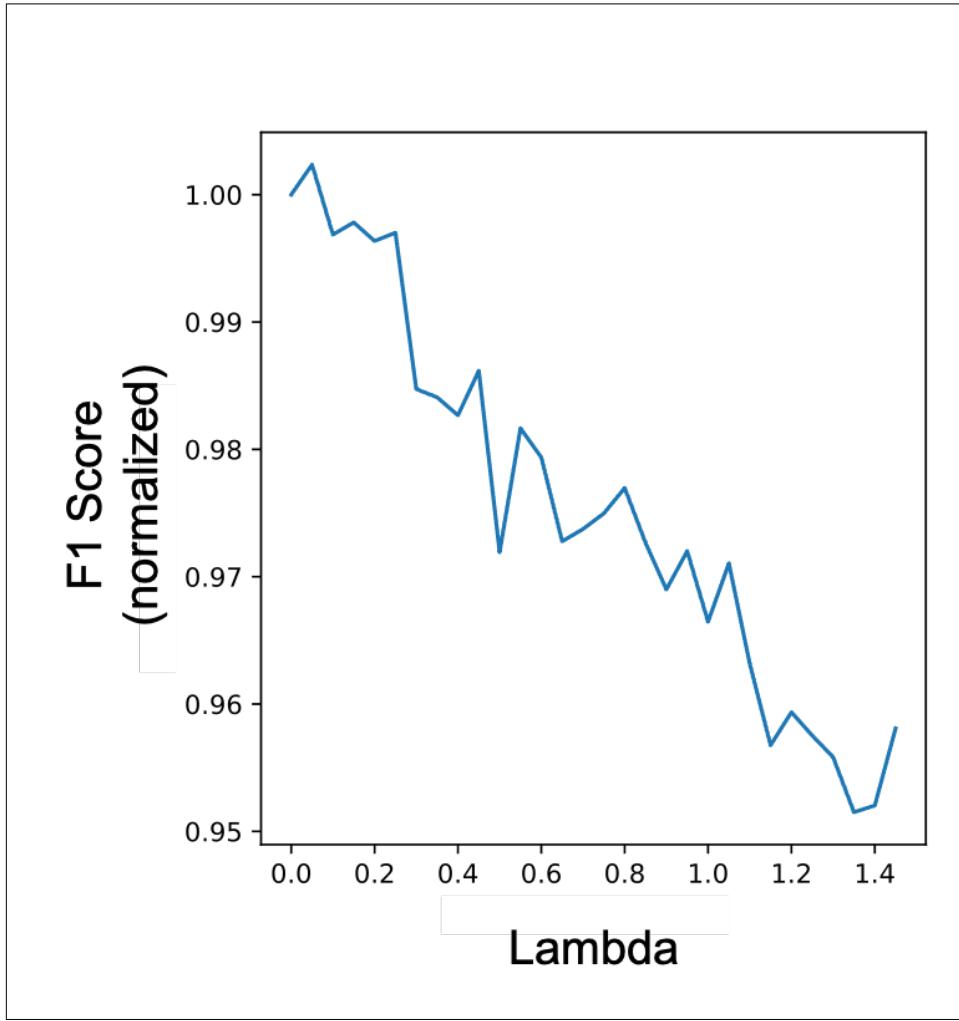


Figure 8-3. F1 scores of the model as the strength of the reweighing scheme is increased.

The results in [Figure 8-4](#) show that increasing `lambda` past 0.8 does not yield meaningful improvements in Black and Hispanic AIRs. Looking back at [Figure 8-3](#), this means we would experience a roughly 3% drop *in silico*. If we were thinking about deploying this model, we'd choose that hyperparameter value for retraining. The

compelling story told between [Figure 8-3](#) and [Figure 8-4](#) is this: simply by applying sampling weights to our dataset so as to emphasize favorable Black and Hispanic borrowers, we can increase AIRs for these two groups, while realizing only a nominal performance drop.

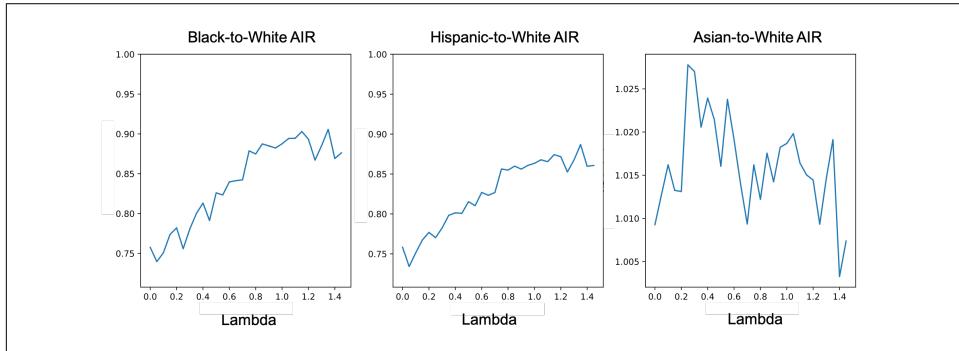


Figure 8-4. Adverse impact ratios of the model as the strength of the reweighing scheme is increased.

Like nearly everything else in ML, bias remediation and our chosen approaches are an experiment, not rote engineering. They're not guaranteed to work and we always need to check if they actually work, first in validation and test data, then in the real world. It's really important to remember that we don't know how this model is going to perform in terms of accuracy or bias once it's deployed. We always hope our *in silico* validation and test assessments are correlated to real-world performance, but there are simply no guarantees. We'd have high hopes that what looks like a ~5% decrease in *in silico* performance, washes out once the model is deployed due to drifting data, changes in real-world operating contexts, and other *in vivo* surprises. All of this points to the need for monitoring both performance and bias once a model is deployed.

Reweighting is just one example of a pre-processing technique, and there are several other popular approaches. Pre-processing is simple, direct and intuitive. As we've just seen, it can result in meaningful improvements in model bias with acceptable accuracy tradeoffs. Check out [AIF 360](#) for examples of other credible preprocessing techniques.

In-processing

Next we'll try an in-processing bias remediation technique. Many interesting techniques have been proposed in recent years, including some that use adversarial models, as in [Mitigating Unwanted Biases with Adversarial Learning](#) or [Fair Adversarial Gradient Tree Boosting](#). The idea behind these adversarial in-processing approaches is straightforward. When an adversarial model cannot predict demographic group

membership from our main model’s predictions, then we feel good that our predictions do not encode too much bias. As highlighted earlier in the chapter, adversarial models also help to capture local information about bias. The rows for which the adversary model is most accurate are likely the rows that encode the most demographic information. These rows can help us uncover individuals who may be experiencing the most bias, complex proxies involving several input features, and other local bias patterns.

There are also in-processing de-biasing techniques that use only one model, and since they are usually a little easier to implement, we’ll focus on one of those for our use case. As opposed to using a second model, these in-processing methods use a dual objective function with a regularization approach. For example, [A Convex Framework for Fair Regression](#) puts forward various regularizers that can be paired with linear and logistic regression models to decrease bias against groups and individuals. [Learning Fair Representations](#) also includes a bias measurement in model objective functions, but then tries to create a new representation of training data that encodes less bias.

While these two approaches focus mostly on simple models, i.e., linear regression, logistic regression, and naïve Bayes, we want to work with trees, and in particular, XGBoost. Turns out, we’re not the only ones. A research group at American Express recently released [FairXGBoost: Fairness-aware Classification in XGBoost](#) which includes instructions and experimental results on introducing a bias regularization term into XGBoost models, using XGBoost’s pre-existing capability to train with custom-coded objective functions. This is how we’ll do in-processing, and as you’ll see soon, it’s remarkably direct to implement and gives good results on our example data.



Before we jump into the more technical descriptions, code and results, we should mention that a great deal of the fairness regularization work we’ve discussed is based on, or otherwise related to, the seminal paper by Kamishima et al., [Fairness-Aware Classifier with Prejudice Remover Regularizer](#).

How does our chosen approach work? Objective functions are used to measure error during model training, where an optimization procedure tries to minimize that error and find the best model parameters. The basic idea of in-processing regularization techniques is to include a measure of bias in the model’s overall objective function. When the optimization function is used to calculate error and the ML optimization process tries to minimize that error, this also tends to result in decreasing measured bias. Another twist on this idea is to use a factor on the bias measurement term within the objective function, or a *regularization hyperparameter*, so that the effect of bias remediation can be tuned. In case readers didn’t know already, XGBoost sup-

ports a wide variety of objective functions so that we can ensure the way error is measured actually maps to the real-world problem at hand. It also supports fully **customized objective functions** coded by users.

The first step in implementing our in-processing approach will be to code a sample objective function. In the code snippet below we define a simple objective function that tells XGBoost how to generate scores, how to:

- calculate the first derivative of the objective function w.r.t model output (gradient, `grad`).
- calculate the second derivative of the objective function w.r.t model output (Hessian, `hess`).
- incorporate demographic information (`protected`) into the objective function.
- control the strength of the regularization with a new parameter (`lambda`, `lambda`).

We also create a simple wrapper for the objective that allows us to specify which groups we want to consider the protected class — those who we want to experience less bias due to regularization, and the strength of the regularization. While simplistic, the wrapper buys us quite a lot of functionality. It enables us to include multiple demographic groups into the protected group. This is important because models often exhibit bias against more than one group, and simply trying to remediate bias for one group may make things worse for other groups. The ability to supply custom `lambda` values is great because it allows for us to tune the strength of our regularization. As shown in the previous section on reweighing, the ability to tune the regularization hyperparameter is crucial for finding an ideal tradeoff with model accuracy.

That's a lot to pack into roughly 15 lines of Python code, but that's why we picked this approach. It takes advantage of niceties in the XGBoost framework, it's pretty simple, and it appears to increase AIR for historically marginalized minority groups in our example data.

```
def make_fair_objective(protected, lambda):
    def fair_objective(pred, dtrain):
        # Fairness-aware cross-entropy loss objective function

        label = dtrain.get_label()
        pred = 1. / (1. + np.exp(-pred))
        grad = (pred - label) - lambda * (pred - protected)
        hess = (1. - lambda) * pred * (1. - pred)

        return grad, hess
    return fair_objective

protected = np.where((train['RACE'] == 'hispanic') | (train['RACE'] ==
```

```
'black'), 1, 0)
fair_objective = make_fair_objective(protected, lambda=0.2)
```

Once that custom objective is defined, we just need to use the `obj=` argument to pass it to XGBoost's `train()` function. If we've written the code correctly, XGBoost's robust training and optimization mechanisms should take care of the rest. Note how little code it takes to train with our custom objective below.

```
model_regularized = xgb.train(params,
                               dtrain,
                               num_boost_round=100,
                               evals=watchlist,
                               early_stopping_rounds=10,
                               verbose_eval=False,
                               obj=fair_objective)
```

Validation and test results for in-processing remediation are available in [Figure 8-5](#) and [Figure 8-6](#). To validate our hypothesis we took advantage of our wrapper function and trained many different models with many different settings of `lambda`. In [Figure 8-6](#), we can see that increasing `lambda` does decrease bias, as measured by increasing Black and Hispanic AIR, while Asian AIR remains roughly constant around the good value of 1. We can increase AIR for the groups we tend to be most concerned about in consumer finance, without engaging in potential discrimination on other demographic groups. That is the result we want to see!

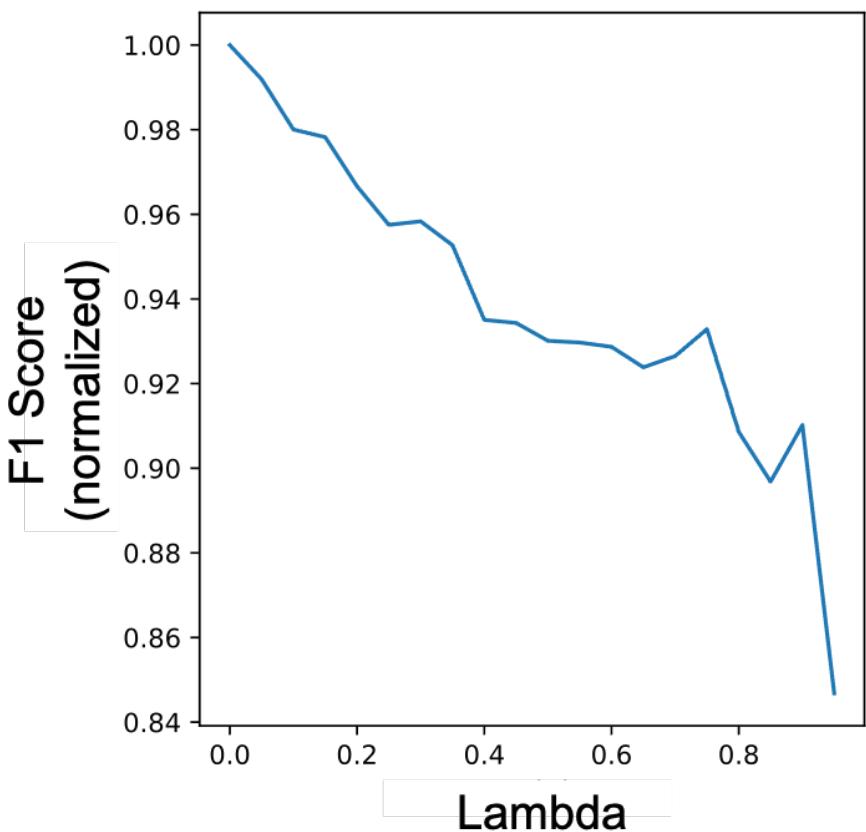


Figure 8-5. The F1 scores of the model as λ is increased.

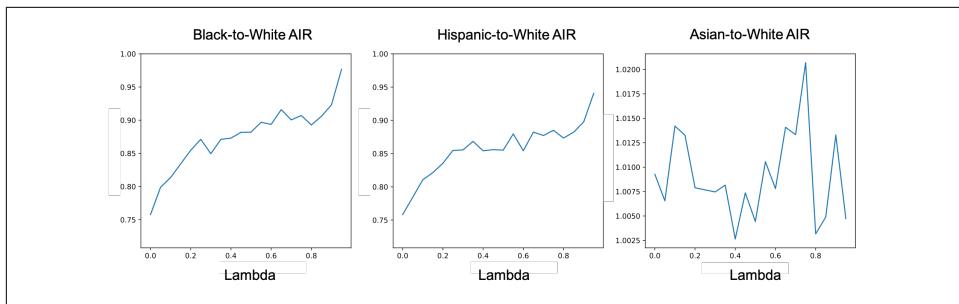


Figure 8-6. AIR values across demographic groups as the regularization factor, `lambda`, is increased.

What about the trade-off between performance and decreased bias? What we saw here is pretty typical in our experience. There's a range of `lambda` values above which Black and Hispanic AIRs do not meaningfully increase, but the F1 score of the model continues to decrease to below 90% the performance of the original model. We probably wouldn't use the model were `lambda` is cranked up to the maximum level, so we're probably looking at a small decrease in *in silico* test data performance and an as yet unknown change in *in vivo* performance.

Post-processing

Next we'll move on to a post-processing techniques. Remember that post-processing techniques are applied after a model has already been trained, so in this section we'll modify the output probabilities of the original model that we trained at the beginning of the chapter.

The technique that we'll apply is called *reject option* post-processing, and it dates back to a 2012 paper by Kamiran et al. Remember that our model has a cutoff value, where scores above this value are given a binary outcome of 1 (an undesirable result for our credit applicants), and scores below the cutoff are given a predicted outcome of 0 (a favorable outcome). Reject option post-processing works on the idea that for model scores *near* the cutoff value, the model is uncertain about the correct outcome. What we do is group together all observations that receive a score within a narrow interval around the cutoff, and then we reassign outcomes for these observations in order to increase the equity of model outcomes. Reject option post-processing is easy to interpret and implement - we were able to do so with another 15-line function.

```
def reject_option_classification(dataset, y_hat, demo_name, protected_groups,
                                 reference_group,
                                 cutoff, uncertainty_region_size):

    # In an uncertainty region around the decision cutoff value, flip protected
    # group predictions
    # to the favorable decision and reference group predictions to the unfavora
```

ble decision.

```
new_predictions = dataset[y_hat].values.copy()

uncertain = np.where(np.abs(dataset[y_hat] - cutoff) <= uncertainty_region_size, 1, 0)
uncertain_protected = np.where(uncertain & dataset[demo_name].isin(protected_groups), 1, 0)
uncertain_reference = np.where(uncertain & (dataset[demo_name] == reference_group), 1, 0)

eps = 1e-3

new_predictions = np.where(uncertain_protected,
                           cutoff - uncertainty_region_size - eps,
                           new_predictions)
new_predictions = np.where(uncertain_reference,
                           cutoff + uncertainty_region_size + eps,
                           new_predictions)

return new_predictions
```

In Figure 8-7 we can see the technique in action. The histograms show the distribution of model scores for each racial group, both before and after the post-processing. We can see that in a small neighborhood of scores around 0.26 (the original model cutoff), we have post-processed all Black and Hispanic people into favorable outcome by assigning them a score at the bottom of the range. Meanwhile, we have assigned White people in this *uncertainty zone* an unfavorable model outcome and left Asian scores unchanged. With these new scores in hand, let's investigate how this technique affects model accuracy and AIRs.

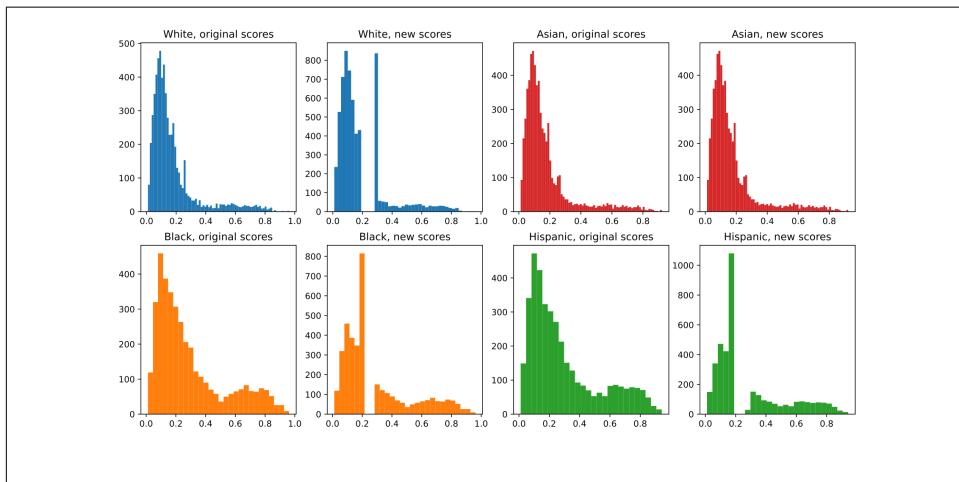


Figure 8-7. Model scores for each demographic group before and after the application of reject option post-processing.

Table 8-7. Original and post-processed F1 scores and adverse impact ratios on validation data.

Model	F1 Score	Black AIR	Hispanic AIR	Asian AIR
Original	0.574	0.736	0.736	1.012
Post-processed	0.541	0.923	0.902	1.06

The results of this experiment are exactly what we would have hoped - we were able to improve Black and Hispanic AIRs above 0.9, while leaving Asian AIRs around 1.00. The price we had to pay in terms of F1 score was a 6% decrease. We don't find this to be a meaningful drop, but if we were concerned we could decrease the size of the uncertainty zone to find a more favorable tradeoff.

Model Selection

The final technique we'll discuss is fairness-aware model selection. To be exact, we'll conduct simple feature selection and random hyperparameter tuning while keeping track of model performance and AIRs. Readers are almost certainly already performing these steps when it comes to performance assessment, so this technique has fairly low overhead costs. Another advantage of model selection as a remediation technique is that it raises the fewest disparate treatment concerns. (On the other end of the spectrum is the reject option post-processing in the previous section, where we literally changed model outcomes depending on the protected group status of each observation.)



Random searches across feature sets and hyperparameter settings often reveal models with improved fairness characteristics and similar performance to a baseline model.

In this section, we'll track F1 and AUC score as our notion of model performance quality. In our experience, evaluating models on multiple measures of quality increases the likelihood of good *in vivo* performance. Another advantage of computing both F1 and AUC scores is that the first is measured on model outcomes and the second uses only output probabilities. If in the future we wanted to change the decision cut-off of our model, or pass the model scores as inputs into another process, we will be glad that we tracked AUC.

One more note before we dive into model selection — model selection is much more than just feature selection and hyperparameter tuning. It can also mean choosing between competing model architectures, or choosing between different bias remediation techniques. In the conclusion of this chapter, we'll round up all of our results to

prepare for a final model selection, but in this section we'll just focus on features and hyperparameters.

In our experience, feature selection can be a powerful remediation technique, but it works best when guided by subject matter experts and when alternative sources of data are available. For example, a compliance expert at a bank may know that a feature in a lending model can be swapped out with an alternative feature that encodes less historical bias. We don't have the luxury of accessing these alternative features, so for our example data we'll only have the option of *dropping* features from our model. In lieu of deep subject matter expertise, we can take advantage of our adversarial models that we already trained. Recall that the adversarial model attempts to predict protected group status using the original set of features. In the code snippet below, we use the adversarial model to select a list of five features that we should consider dropping in a random grid search:

```
adversarial_shap_values = pd.DataFrame(  
    shap.TreeExplainer(adversary_model).shap_values(explanation_data[features]),  
    columns=features)  
features_to_drop =\  
    list(adversarial_shap_values.abs().mean().sort_values(ascending=False).head(5).index)
```



By using XAI techniques and adversarial models in tandem, we can learn a lot about which features are driving disparities in our model and the value of alternative features.

Next, we'll train five new models using the original hyperparameters and dropping these five features. Between feature selection and hyperparameter tuning, we're about to train a lot of different models, so we'll employ five-fold cross-validation using our original training data. If we chose the variant with the best performance on validation data, we run an increased risk of selecting a model that performs best only due to random chance.



Any time we evaluate multiple models on the same dataset, be careful about overfitting and multiple comparisons. Employ best practices such as reusable holdout, cross validation, boot strapping, out-of-time holdout data, and post-deployment monitoring to ensure that our results generalize.



While the Rashomon Effect may mean we have many good models to chose from, we should not forget that may also be a sign of instability in our original model. If there are many models with settings similar to our original model, that also perform differently from our original model, this points to under- and mis-specification issues. Remediated models must also be tested for stability, safety, and performance issues. See Chapter's [unique_chapter_id_3], [unique_chapter_id_7] and [unique_chapter_id_9] for more information.

After training these new models using cross-validation, we were able to realize an increase in Black and Hispanic cross-validation AIRs, alongside a small decrease in model cross-validation AUC. The most offending feature was PAY_AMT5, so we'll proceed with random hyperparameter tuning without this feature.

To choose new model hyperparameters, we'll use random grid search using the `sklearn` API. Since we want to cross-validation AIRs throughout this process, we had to put together a scoring function to pass into `sklearn`. To simplify the code, we only track Black AIR below — since it has been correlated to Hispanic AIR throughout our analysis — but an average measure of AIR across protected groups is likely preferable. The code snippet shows how we used global variables and the `make_scorer()` interface to get this done:

```
fold_number = -1

def black_air(y_true, y_pred):

    global fold_number
    fold_number = (fold_number + 1) % num_cv_folds

    model_metrics = perf_metrics(y_true, y_score=y_pred)
    best_cut = model_metrics.loc[model_metrics['f1'].idxmax(), 'cutoff']

    data = pd.DataFrame({'RACE': test_groups[fold_number],
                         'y_true': y_true,
                         'y_pred': y_pred},
                         index=np.arange(len(y_pred)))

    disparity_table = fair_lending_disparity(data, y='y_true', yhat='y_pred',
                                              demo_name='RACE', groups=race_levels, reference_group='white',
                                              cutoff=best_cut)

    return disparity_table.loc['black']['AIR']

scoring = {
    'AUC': 'roc_auc',
    'Black AIR': sklearn.metrics.make_scorer(black_air, needs_proba=True)
}
```

Next, we defined a reasonable grid of hyperparameters and built 50 new models:

```
parameter_distributions = {
    'n_estimators': np.arange(10, 221, 30),
    'max_depth': [3, 4, 5, 6, 7],
    'learning_rate': stats.uniform(0.01, 0.1),
    'subsample': stats.uniform(0.7, 0.3),
    'colsample_bytree': stats.uniform(0.5, 1),
    'reg_lambda': stats.uniform(0.1, 50),
    'monotone_constraints': [new_monotone_constraints],
    'base_score': [params['base_score']]}
}

grid_search = sklearn.model_selection.RandomizedSearchCV(xgb.XGBClassifier(
    random_state=12345,
    use_label_encoder=False,
    eval_metric='logloss'),
    parameter_distributions,
    n_iter=50,
    scoring=scoring,
    cv=zip(train_indices,
    test_indices),
    refit=False,
    error_score='raise').\
    fit(train[new_features], train[target].values)
```

The results of our random model selection procedure are shown in [Figure 8-8](#). Each model is a point on the plot, with Black cross-validation AIR values on the x-axis and cross-validation AUC on the y-axis. As we've done here, it is useful to normalize model accuracy to the baseline value in order to easily make statements like "this alternative model shows a 2% drop in AUC from the original model." Given this distribution of models, how do we go about choosing one for deployment?

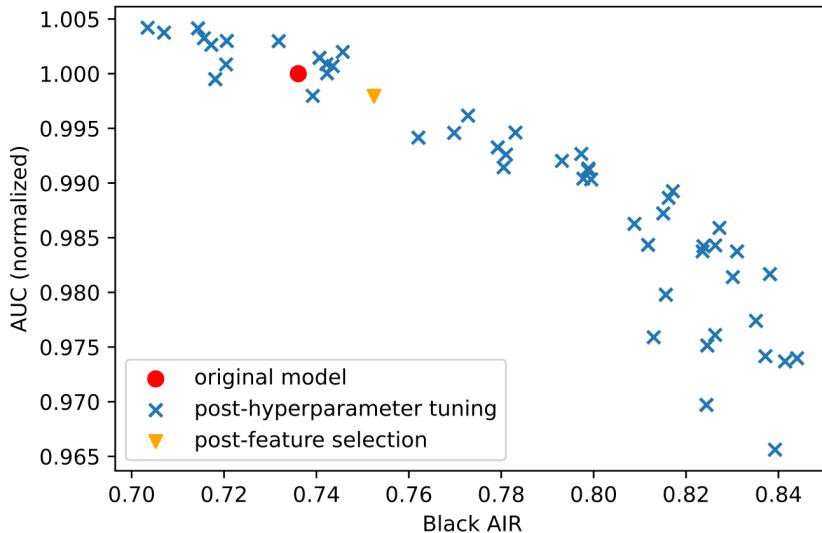


Figure 8-8. The normalized accuracy and Black AIRs of each model after feature selection and hyperparameter tuning.

A common problem with bias remediation approaches is that they often just move bias around from one demographic group to another. For example, women are now favored sometimes in credit and employment decisions in the U.S. It wouldn't be surprising to see a bias remediation technique dramatically decrease favorable outcomes for women to increase them for other minorities, but that's not the outcome anyone really wants. If one group is disproportionately favored, and bias remediation equals that out — great. If, on the other hand, one group is favored a bit, and bias remediation ends up harming them to increase AIR or other statistics for other groups, that's obviously not great. In the next section, we'll see how these two alternative models stack up against the other remediation techniques applied in this chapter.

Conclusion

In [Table 8-8](#), we've aggregated the results for all of the models trained in this chapter. We chose to focus on two measures of model accuracy - F1 score and AUC - and two measures of model bias - AIRs and false positive rate disparity.

Table 8-8. Comparison on test data between bias remediation techniques.

Measurement	Original Model	Pre-processing (reweighting, $\lambda = 0.2$)	In-processing (regularized, $\lambda = 0.2$)	Post-processing (reject option, window = 0.1)	Model Selection
AUC	0.798021	0.774183	0.764005	0.794894	0.789016
F1	0.558874	0.543758	0.515971	0.533964	0.543147
Asian AIR	1.012274	1.010014	1.001185	1.107676	1.007365
Black AIR	0.735836	0.877673	0.851499	0.901386	0.811854
Hispanic AIR	0.736394	0.861252	0.851045	0.882538	0.805121
Asian FPR Disparity	0.872567	0.929948	0.986472	0.575248	0.942973
Black FPR Disparity	1.783528	0.956640	1.141044	0.852034	1.355846
Hispanic FPR Disparity	1.696062	0.899065	1.000040	0.786195	1.253355

The results are exciting: many remediation techniques tested are able to realize meaningful improvements in AIRs and FPR disparities for Black and Hispanic borrowers with no serious negative impact on Asian AIR. This is possible with only marginal changes in model performance.

How should we choose which remediation technique to apply to our high-risk model? Hopefully this chapter has convinced readers to try many things. Ultimately, the final decision rests with the law, business decisions, and the diverse team of stakeholders that we assemble. If we're working in a traditionally regulated vertical, we can really only choose from model selection options today. If we're outside of the verticals where disparate treatment is strictly prohibited, we have a much wider selection of remediation strategies to choose from. (And don't forget the bias remediation [decision tree](#) - slide 40.) We'd likely pick the preprocessing option for remediation given the minimal decrease in model performance versus in-processing and because post-processing knocks some performance disparities out of acceptable ranges.

Whether we are using model selection as a bias mitigation technique or whether we have different pre-, in-, and post-processed models to choose from, a rule of thumb for picking a remediated model is to:

1. Reduce the set of models to those that perform sufficiently well to meet business needs, e.g., performance within 5% of the original model.
2. Among those models, pick the one that most closely:
 - remediates bias across all originally disfavored groups, e.g., all disfavored groups AIR increased to ≥ 0.8
 - does not discriminate against any initially favored group, e.g., no originally favored groups AIR decreased to < 0.8

3. Consult business partners, legal and compliance experts, and diverse stakeholders as part of the selection process.

If we're training a model has the potential to impact people — and most models do — we have an ethical obligation to test it for bias. And when we find bias, we need to mitigate or remediate it. What we've gone over in [Chapter 8](#) is the technical part of bias management processes. To get bias remediation right, also involves extending our release timelines, lots of careful communication between different stakeholders, and lots of re-training and re-testing of ML models and pipelines. We're confident that if we slow down, ask for help and input from stakeholders, and apply the scientific method, we will be able to tackle real-world bias challenges and deploy performant and minimally biased models.

Resources

Code Examples:

- [Machine-Learning-for-High-Risk-Applications-Book](#)

Tools for Managing Bias:

- [aequitas](#)
- AI Fairness 360:
 - [Python](#)
 - [R](#)
- [algofairness](#)
- [fairlearn](#)
- [fairml](#)
- [fairmodels](#)
- [fairness](#)
- [solas-ai-disparity](#)
- [tensorflow/fairness-indicators](#)
- [Themis](#)

Red-teaming XGBoost

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

In Chapter 5 we introduced a number of concepts related to the security of machine learning (ML) models. Now we will put them to into practice. In Chapter 9, we’ll show you how to hack your own models so that you can add red-teaming into to your model debugging repertoire. The main idea of the chapter is when you know what hackers will try to do to your model, then you can try it out first and devise effective defenses. We’ll start out with a concept refresher that reintroduces common ML attacks and countermeasures, then we’ll dive into examples of attacking an XGBoost classifier trained on structured data. (You can find examples of attacks on computer vision models all over the internet, but the tutorials associated with [cleverhans](#) are a great place to start.) We’ll then introduce two XGBoost models, one trained with the standard black-box approach, and one trained with constraints and a high degree of L2 regularization. We’ll use these two models to explain the attacks and to test whether transparency and L2 regularization are adequate countermeasures. After that, we’ll jump into attacks that are likely to be performed by external adversaries against a black-box ML API: model extraction and adversarial example attacks. From

there, we'll try out insider attacks that involve making deliberate changes to an ML modeling pipeline: data poisoning and model backdoors. Let's get started — remember to bring your tinfoil hat, and your adversarial mindset from [Chapter 5](#).

Concept Refresher

It's worth reminding ourselves why we are interested in ML model attacks. ML models can hurt people and be hurt — manipulated, altered, destroyed — by people. Broadly speaking, security incidents are a major way that operators, users, and the general public are harmed by technology. Bad actors may seek to induce beneficial outcomes for themselves or harmful outcomes for others, they may commit corporate espionage, steal intellectual property and steal data. We don't want our ML models to be a sitting duck for that kind of malicious activity! In [Chapter 5](#), we called this way of thinking the *adversarial mindset*. While your ML model might be your perfect Python baby that's also primed to make your organization millions of dollars, it's also a legal liability, a security vulnerability, and an end point for hackers to explore. There's no way around this reality, especially for important high-impact public-facing ML systems. Let's not be naïve. Let's do the hard work required to check that our model isn't full of vulnerabilities that can leak training data, leak models themselves, or allow bad actors to trick our systems out of money, intellectual property, or worse. Now, let's refresh our memory on some of those [Chapter 5](#) concepts, attacks and countermeasures!

CIA Triad

Recall that, broadly, we break information security incidents down into three categories defined by the CIA triad: confidentiality, integrity, and availability attacks.

- **Confidentiality Attacks:** violate the confidentiality of some data associated with an ML model, typically the logic of the model or the model's training data. Model extraction attacks expose the model, while membership inference attacks expose the training data.
- **Integrity Attacks:** compromise the behavior of the model, typically to alter predictions in ways that are beneficial to the attacker. Adversarial example, data poisoning, and backdoor attacks all violate the integrity of a model.
- **Availability Attacks:** prevent a user of the model from accessing it in a timely or serviceable fashion. In ML, some have described algorithmic discrimination as a form of availability attack since minority groups don't receive the same service from the model as majority groups. However, most availability attacks will be general denial-of-service attacks targeted at the service running the model, and not specialized for ML. We won't try an availability attack, but you should check with your information technology (IT) partners and make sure your public-

facing ML models have standard countermeasures in place to mitigate availability attacks.

With that brief reminder of the CIA triad, let's turn to more details about each of our planned red-teaming attacks.

Attacks

We'll roughly group ML attacks under two main categories for the concept refresher: external attacks and insider attacks. External attacks are defined as the attacks that an external adversary would be most likely to try on our model. The setup for these attacks is that we've deployed the model as an API, but perhaps been a little sloppy when it came to security. We're going to assume that we can interact with the model as a black-box, do so anonymously, and that we can have a reasonable number of data submission interactions with the model. Under these conditions an external attacker could extract the basic logic of our model in a model extraction attack. With or without that blueprint, but it's easier and more harmful with it, the attacker could then begin to craft adversarial examples that look like normal data, but evoke surprising results from the model. With the right adversarial examples, an attacker can play our model like a fiddle. If the hackers are successful in conducting the two previous attacks, they might become more bold, and try perhaps a more sophisticated and harmful attack: membership inference. Let's look at the different types of external attacks in a bit more detail:

- **Model Extraction** is a confidentiality attack, meaning it compromises the confidentiality of an ML model. To conduct a model extraction attack, a hacker submits data to a prediction API, gets predictions back and builds a surrogate model between the submitted data and the received predictions to reverse-engineer a copy of the model. With this information, they may uncover proprietary business processes and decision-making logic. The extracted model also provides a great test bed for subsequent attacks.
- **Adversarial Examples** is an integrity attack. It compromises the correctness of model predictions. To perform an adversarial example attack a hacker will probe how a model responds to input data. In computer vision systems, gradient information is often used to fine tune images that evoke strange responses from the model. For structured data, we can use individual conditional expectation (ICE) or genetic algorithms to find rows of data that cause unexpected model predictions.
- **Membership Inference** is a confidentiality attack that seeks to compromise model training data. It's a complex attack that requires two models. The first is a surrogate model similar to those that would be trained in a model extraction attack. The second stage model is then trained to decide whether a row of data is

in the training data of the surrogate model or not. When that second-stage model is applied to a row of data, it can decide whether that row was in the training data of the surrogate model or not, and can often extrapolate to decide whether that row was also in the original model training data!

Now for those insider attacks. Sadly, we can't always trust our fellow employees, consultants, or contractors. And worse yet, people can be extorted into committing bad acts, whether they want to or not. In a data poisoning attack, someone changes training data in a way that allows them, or their associates, to manipulate the model later. In a backdoor attack, someone alters the scoring code of the model so that they can later access the model in unauthorized ways. In both data poisoning and backdoor attacks, it's most likely that the perpetrator would seek to gain financially themselves, and alter the data or scoring code accordingly. However, it's also possible that a bad actor would change an important model in a way that hurt others, and not necessarily to benefit themselves.

- **Data Poisoning** is an integrity attack that changes training data to change future model outcomes. To conduct the attack, someone only needs access to model training data. They try to change the training data in subtle ways that will reliably alter model predictions, in ways they or associates can exploit later when interacting with the model.
- **Backdoors** are an integrity attack that change a model's scoring (or inference) code. The goal of a backdoor attack is to introduce new branches of code into the complex tangle of coefficients and if-then rules that is a deployed ML model. Once the new branch of code has been injected into the scoring engine, it can be exploited later by those who know how to trigger it, e.g., by submitting unrealistic combinations of data into a prediction API.

We didn't go back over evasion and impersonation attacks, but they are covered in the case in [Chapter 5](#). In our research, evasion and impersonation attacks are the most common kinds of attacks today. They're typically applied to ML-enhanced security, filtering or payment systems. In computer vision, they usually involve some kind of physical manipulation of an ML system, for instance wearing a realistic mask or camouflaging yourself. For structured data, these attacks just mean altering a row of data to have similar values (impersonation), or dissimilar values (evasion), when compared to some user of a model. We're confident that if you think through impersonation and evasion, you'll come up with solid ways to red-team them. Keep in mind that evading fraud detection ML models is a long-running cat and mouse game between fraudsters and financial institutions, and that's probably the most likely place you'd run into evasion attacks based on manipulating structured data.

Countermeasures

Most ML attacks are premised on ML models being overly-complex, unstable, overfit, black-boxes. The overly-complex black box is important because humans will have a hard time understanding if an uber-complex system is being manipulated. Instability is important for attacks because it leads to scenarios where minor perturbations to input data can lead to dramatic and unexpected changes in model outputs. Overfitting results in unstable models, and comes into play for membership inference attacks. If a model is overfit, it behaves quite differently on new data than on training data, and we can use that performance differential to infer if a row of data was used to train the model. With all this in mind, we're going to try two simple counter measures.

- **L2 regularization** is a penalty placed on the squared sum of model coefficients in the model's error function, or some other measure of model complexity. Strong L2 regularization prevents any one coefficient, rule, or interaction from becoming too large and important in the model. If no single feature or interaction is driving the model, it's harder to construct adversarial examples. L2 regularization tends to make all model coefficients smaller as well, making model predictions more stable and less subject to wild swings. L2 regularization is also known to improve models' generalization capabilities, which should also help to counter membership inference attacks.
- **Monotonicity Constraints** make the model more stable and interpretable, both of which are general mitigants of ML attacks. If a model is highly interpretable, this changes its entire security profile. We know how the model should behave and can more easily identify when it is manipulated. Confidentiality attacks lose their bite, because everyone knows how the model works when it obeys reality. If the constraints prevent the model from generating surprising predictions, then there's really no way to conduct an adversarial example attack. If constraints enforce realistic behavior on the model, then data poisoning should be less effective. Constraints should also help with generalization, making membership inference more difficult.

We also hope there is some synergy between these two general countermeasures. Both L2 regularization and constraints increase the stability of the model. By using them, we are trying to ensure we won't see big changes in model outputs based on small changes to model inputs. With constraints in particular, we are also making sure our model simply can't surprise us. The constraints mean it has to obey obvious, causal reality, and hopefully adversarial examples will be much more difficult to find and data poisoning will be less damaging. Both should also decrease overfitting, and provide some defense against membership inference.

Other important countermeasures include **throttling**, **authentication**, **robust ML** and **differential privacy approaches**. Throttling slows down predictions if someone interacts with an API too frequently or in a strange way. Authentication prevents anonymous use, which should generally disincentivize attacks. Robust ML approaches create models that are custom-designed to be more robust to adversarial examples and data poisoning. Differential privacy methodically corrupts training data to obscure it if a model extraction or membership inference attack occurs. We'll be using L2 regularization as a more accessible alternative to robust ML and differential privacy approaches. We've explained that L2 regularization acts to create more stable models above, but you may need a reminder L2 regularization is equivalent to Gaussian noise injection in training data! There's no guarantee this works as well as real differential privacy methods, but we'll be testing how well it actually works in the code examples. Now that we've gone back over the main technical points, let's train some XGBoost models.

Model Training

In our example models, we'll be deciding whether to extend an API user an increased line of credit. You may be thinking that a credit model would be one of the most well-protected models out there, and you're right. But similar ML models are used in the fintech and crypto wild west, and if you think just because a computer technology is deployed at a big bank then it's safe, bank regulators may have some **thoughts** for you. We'll introduce plausible attack scenarios with each example, but the reality is that real-world attacks can be strange and surprising, and can happen to any model.

In all our attacks, we're going to try to hack two different models. You'd likely only red-team the model you have planned for deployment. But we're going to try an experiment in this chapter. The first model will be a typical XGBoost model, unconstrained and somewhat overfit with little regularization beyond that provided by column and row sampling. We expect this model will be easier to hack due to overfitting and instability. We set `max_depth` to 10 in an effort to overfit and we specify the other hyperparameters as follows:

```
params = {"ntrees": 100,  
          "max_depth": 10,  
          "learn_rate": 0.1,  
          "sample_rate": 0.9,  
          "col_sample_rate_per_tree": 1,  
          "min_rows": 5,  
          "seed": SEED,  
          "score_tree_interval": 10  
      }
```

We train our typical XGBoost model with no frills:

```
xgb_clf = H2OXGBoostEstimator(**params)
xgb_clf.train(x=features, y=target, training_frame=training_frame, validation_frame=validation_frame)
```

Before we get to far into model training, note that we'll be using the `h2o` interface to XGBoost, specifically so that we can generate Java scoring code and try a backdoor attack on that code later. That also means the hyperparameter names might be a little different than when using native XGBoost.

For the model we hope will be more robust, we first determine monotonicity constraints using Spearman correlation, just like in [Chapter 6](#). These constraints have two purposes, both based on the commonsense transparency they provide. First, they should keep the model more stable under an integrity attack. Second, they should make a confidentiality attack less worth it for an attacker. A constrained model is going to be more difficult to manipulate because its logic follows predictable patterns, and should not hide too many secrets that could be sold or used for future attacks. Here's how we set up the constraints:

```
corr = pd.DataFrame(train[features + [target]].corr(method='spearman')[target]).iloc[:-1]
corr.columns = ['Spearman Correlation Coefficient']
values = [int(i) for i in np.sign(corr.values)]
mono_constraints = dict(zip(corr.index, values))
mono_constraints
```

The constraints defined by our approach are negative for `BILL_AMT*`, `LIMIT_BAL`, and `PAY_AMT*` features. They are positive for `PAY_*` features. These constraints are intuitive. As bill amounts, credit limits and payment amounts get larger, the probability of default from our constrained classifier can only decrease. As someone becomes later with their payments, probability of default can only increase. For `h2o` monotonicity constraints need to be defined in a dictionary and they look like this for our model with countermeasures:

```
{'BILL_AMT1': -1, 'BILL_AMT2': -1, 'BILL_AMT3': -1, 'BILL_AMT4': -1,
'BILL_AMT5': -1, 'BILL_AMT6': -1, 'LIMIT_BAL': -1, 'PAY_0': 1, 'PAY_2': 1,
'PAY_3': 1, 'PAY_4': 1, 'PAY_5': 1, 'PAY_6': 1, 'PAY_AMT1': -1, 'PAY_AMT2': -1,
'PAY_AMT3': -1, 'PAY_AMT4': -1, 'PAY_AMT5': -1, 'PAY_AMT6': -1}
```

We also use a grid search to look across a broad set of models in parallel fashion. Because our training data is small, we can afford to do a Cartesian grid search across most important hyperparameters.

```
# settings for XGB grid search parameters
hyper_parameters = {'reg_lambda': [0.01, 0.25, 0.5, 0.99],
'min_child_weight': [1, 5, 10],
'eta': [0.01, 0.05],
'subsample': [0.6, 0.8, 1.0],
'colsample_bytree': [0.6, 0.8, 1.0],
'max_depth': [5, 10, 15]}
```

```

# initialize cartesian grid search
xgb_grid = H2OGridSearch(model=H2OXGBoostEstimator,
                         hyper_params=hyper_parameters,
                         parallelism=3)

# training w/ grid search
xgb_grid.train(x=features,
               y=target,
               training_frame=training_frame,
               validation_frame=validation_frame,
               seed=SEED)

```

Once we locate a set of hyperparameters that don't overfit our data, we then retrain using that set of hyperparameters, `params_best`, and our monotonicity constraints.

```

xgb_best = H2OXGBoostEstimator(**params_best, monotone_constraints=mono_cons)
xgb_best.train(x=features, y=target, training_frame=training_frame, validation_frame=validation_frame)

```

Examining the receiver operating characteristic (ROC) plots for both models, shows we likely achieved our goal of having two different models to red-team. The typical model, on top in Figure Figure 9-1, shows the canonical signs of overfitting. It has high training area under the curve (AUC), and much lower validation AUC. Our constrained model looks much more well-trained at the bottom of Figure Figure 9-1. It has the same validation AUC as the typical model, but a much lower training AUC, indicating much less overfitting. While we can't be certain, the monotonicity constraints probably helped mitigate over fitting.

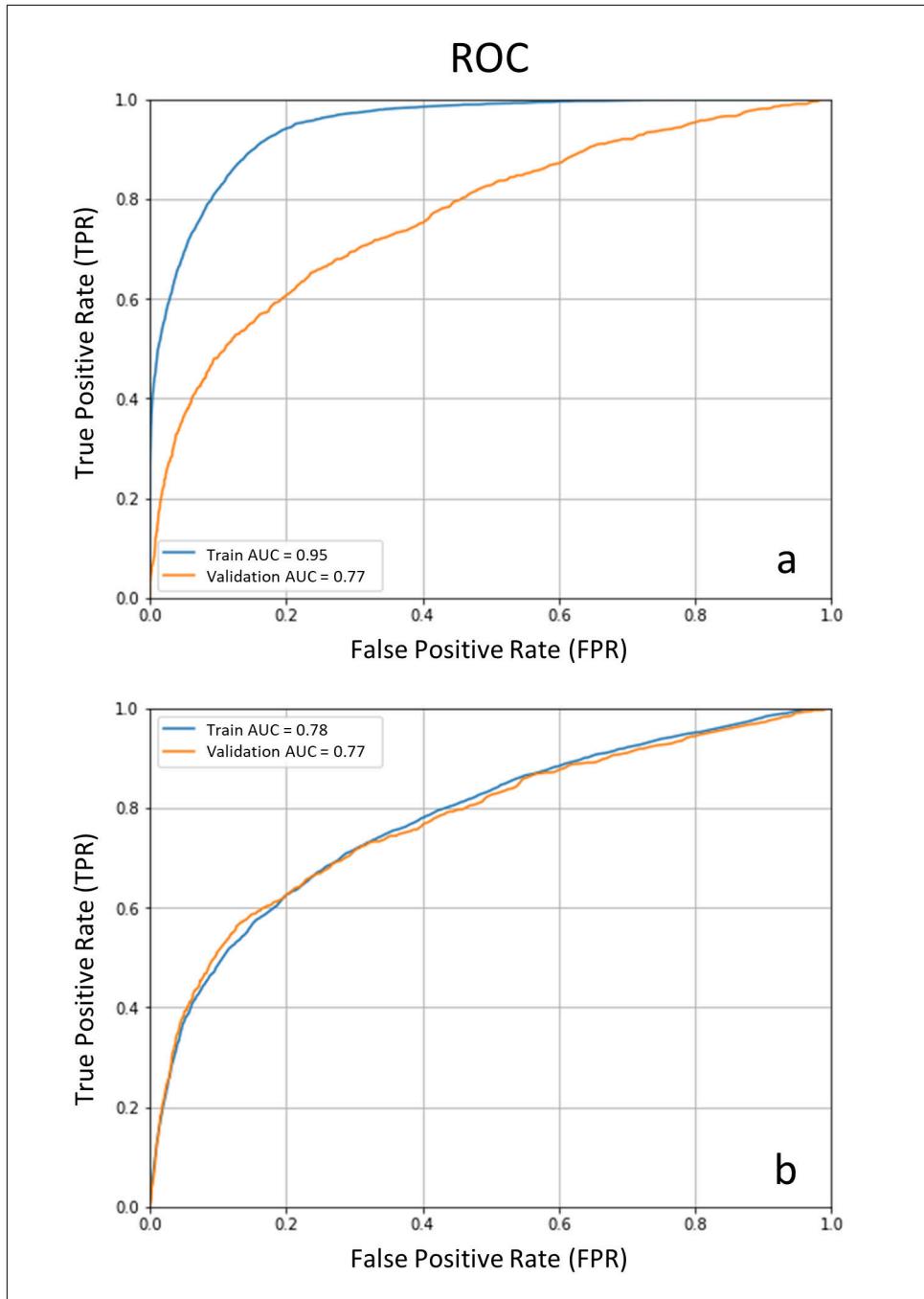


Figure 9-1. ROC curves for a. an overfit XGBoost model, and b. a highly regularized and constrained XGBoost model.

Now that we have two models, we'll proceed with both our experiment and our red-teaming. We'll be looking to confirm what's reported in the literature and what we hypothesize — that the typical model will be easier and more fruitful to attack. It should be unstable, hiding many nonlinearities and high-degree interactions. This makes attacking it more worthwhile. A hacker could likely find aspects of the black-box GBM that could be exploited for attack, using say, adversarial examples. Because it's overfit, the typical model should also be more susceptible to model extraction attacks. Backdoors should be easier too — we'll attempt to hide new code in the tangle of complex if-then rules that defined the overfit GBM.

All of these attacks play on one of the fundamental premises of ML security — a determined attacker can learn more about your overly complicated black-box than you'll ever be motivated to know. The attacker can exploit this information imbalance in many ways. We'll hope our constrained and regularized model is both harder to attack using data poisoning, backdoors, and adversarial examples *and* less useful to try a confidentiality attack on, because anyone with any domain knowledge can guess how it works and know when it's being manipulated.

Attacks for Red-teaming

We'll consider model extraction and adversarial example attacks as more likely to be conducted by someone outside of the organization. We'll red-team for these attacks as if we were external bad actors. We'll treat all interactions with ML models as though we were interacting with a black-box API, but you'll see that we can still learn a lot about a so-called black box. We're also assuming that authentication is not required to access the API and that we can access the API to receive at least a few batches of predictions. Our attacks, when successful, will build off each other. You'll see that the initial model extraction attack is extremely damaging, not only because we can learn a lot about the attacked model and its training data, but because it creates a test bed for attackers to hone future hacks.

Model Extraction Attacks

The basic necessary condition for a model extraction attack is that a bad actor can submit data to a model and receive predictions. As this is the way ML is usually designed to work, model extraction attacks are hard to eradicate completely. More specific scenarios for model extraction include weak authentication requirements, say providing just an email address to create an account to use the API, and that hackers can receive thousands of predictions a day from the API. Another fundamental requirement is for a model to hide some information worth stealing. If a model is highly transparent and well-documented, there's less reasons to extract it explicitly.

Since our model is a credit model, we'll blame a "go fast and break things" culture at a new fintech company that wants to rush their ML-based credit scoring API into pro-

duction in an effort to create hype in their market. We could just as easily blame byzantine security procedures at a major bank that allows, at least for a short time period, a product API to be more accessible than it should be. In either case, model extraction could be conducted by corporate competitors who want to understand your organization's proprietary business rules or by hackers who want free money. None of these scenarios are particularly far-fetched, which begs the question, how many model extraction attacks are occurring right now? Let's get into how to red-team for them so that your organization won't fall victim to one of these attacks.

The starting point for the attack is an API endpoint. We'll set up a basic endpoint as follows:

```
def model_endpoint(observations: pd.DataFrame):  
  
    pred_frame = h2o.H2OFrame(observations)  
    prediction = xgb_clf.predict(pred_frame)[‘p1’].as_data_frame().values  
  
    return prediction
```

From there we submit data to the API endpoint to receive predictions to start the red-teaming exercise. The type of data submitted to the API appears very important in the success of our attack. At first we tried to guess the distributions of the input features and simulate data by drawing from the distributions we assigned to each feature. That didn't work so well, so we applied the *model-based synthesis* approach described in a well-known paper by *Shokri et al.* This method gives more weight to simulated data rows that evoke a high-confidence response from the API endpoint. By combining our best guess at the distributions of the input features and then using the endpoint to check each simulated row of data, we were able to simulate a set of data that is similar enough to the original dataset to attempt several model extraction attacks. The downside of the model-based synthesis approach is that it involves more interactions with the API, hence, more opportunities to get caught.



The success of model extraction attacks appears to depend heavily on good simulation of training data.

With realistic data in-hand, we can now proceed to the attack. We conduct three different model extraction attacks using a decision tree, a random forest, and an XGBoost GBM as the extracted surrogate model. We submitted our simulated data back to the API endpoint, received predictions, and then trained these three models using the simulated data as inputs and the received predictions as the target. XGBoost seemed to make the best copy of the attacked model in terms of accuracy, perhaps because the model behind the endpoint was also an XGBoost GBM. This is what training the extracted XGBoost model looks like:

```

drand_train = xgb.DMatrix(random_train[features],
                          label=model_endpoint(random_train[features]))

drand_valid = xgb.DMatrix(random_valid[features],
                          label=model_endpoint(random_valid[features]))

params = {
    'objective': 'reg:squarederror',
    'eval_metric': 'rmse',
    'eta': 0.1,
    'max_depth': 3,
    'base_score': base_score,
    'seed': SEED
}

watchlist = [(drand_train, 'train'), (drand_valid, 'eval')]

extracted_model_xgb = xgb.train(params,
                                 drand_train,
                                 num_boost_round=15,
                                 evals=watchlist,
                                 early_stopping_rounds=5,
                                 verbose_eval=False)

```

We split our simulated data into `drand_train` training and `drand_valid` validation partitions. For each partition, the target feature comes from the API endpoint. We then apply very simple hyperparameter settings and train the extracted model. A grid search may have lead to a better fit on these simulated rows of data, which may be the attacker's goal on some occasions. We wanted to steal a more generalizable model, and kept our parameterization straightforward. XGBoost was able to achieve an R^2 of 0.635 against the API predictions using the simulated data. Figure [Figure 9-2](#) shows a plot of actual predictions vs. extracted predictions across our simulated training data, simulated test data, and the actual validation data. While no extracted models are a perfect fit for the API predictions, they all show a strong degree correlation to the API predictions. Since this is a red-teaming exercise, we can see our extracted XGBoost achieved an R^2 of 0.632 on the real validation data. This shows just how close we could get our simulated data to the training data of the original model, and that alone is a serious confidentiality risk. We now have some idea what the training data looks like, and the tools to learn a lot more about it.

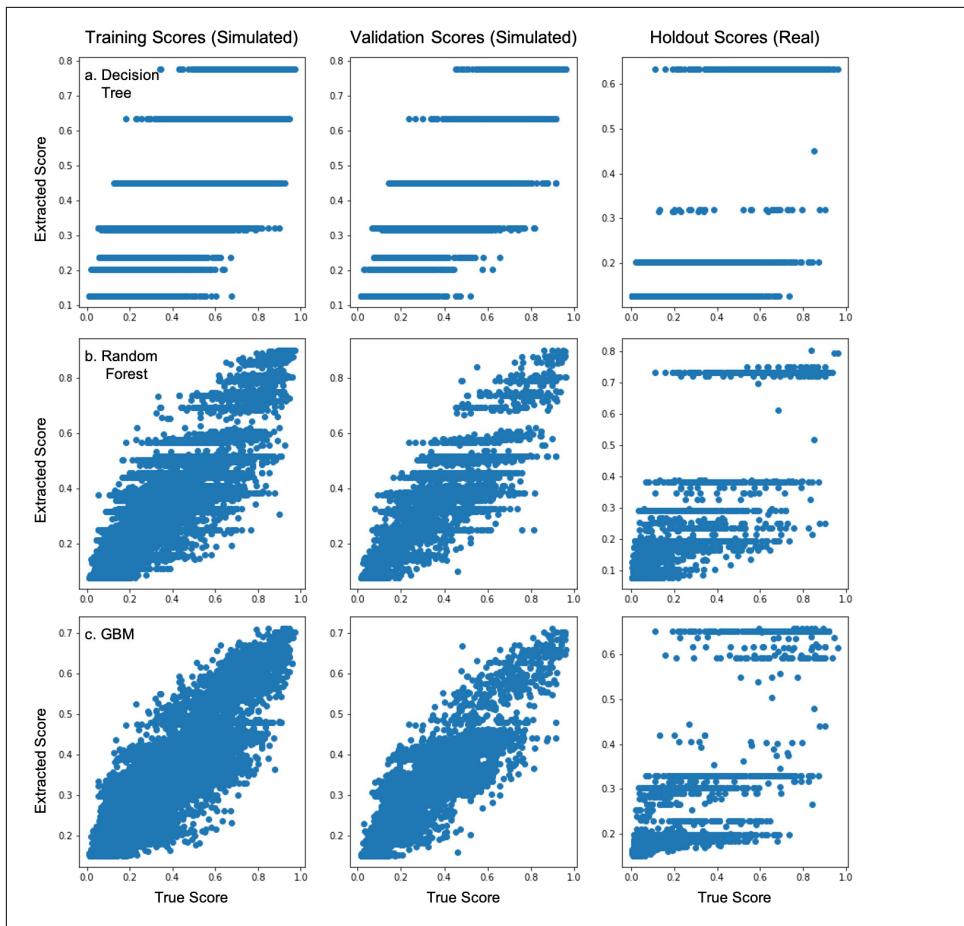


Figure 9-2. A comparison of extracted model scores vs. true model scores across simulated training, simulated test, and real holdout data for a. a decision tree, b. a random forest, and c. a GBM.

An important result to note is that extracting the constrained model worked much better. Where we saw R^2 in the range of 0.6 for the unconstrained model, we saw R^2 in the range of 0.9 for the constrained model. The assumption is that the constrained model would also follow other tenants of risk management, such as thorough documentation. If how a model works is transparent, extracting it shouldn't be worth the effort, but this finding does contravene some of our original hypotheses about the constrained and regularized model.



Constrained models may be much easier to extract from API endpoints. Such models should be accompanied by thorough consumer-facing documentation that undercuts the motivation for an extraction attack.

Being able to extract a model like this is a bad omen for ML security. Not only are we starting to get an idea of what the supposedly confidential training data looks like, but we have a set of extracted models. Each of the extracted models is a compressed representation of the training data and a summary of an organization's business processes. We can use explainable AI (XAI) techniques to torture even more information out of these extracted models. We can use feature importance, Shapley values, partial dependence, individual conditional expectation (ICE), accumulated local effect (ALE) and more to maximize the exfiltration of confidential information. Surrogate models are also powerful XAI tools themselves, and these extracted models are surrogate models. While the decision tree gave the worst numerical accuracy with respect to reproducing the API predictions, it is also highly interpretable. Watch below as we use this model to craft adversarial examples with ease, and do so with fewer interactions with the model API, drawing less attention to our red-teaming efforts.

Adversarial Example Attacks

Adversarial example attacks are likely the first attack that comes to mind for many readers. They have even fewer pre-conditions than model extraction. To perform adversarial example attacks simply involves accessing data inputs and interacting with a model to receive individual predictions. Like model extraction attacks, adversarial example attacks are also premised on the use of black-boxes. However, the perspective is a little different than in the last attack. Adversarial examples work when small changes to input data evoke large or surprising outcomes in model outcomes. This type of nonlinear behavior is a hallmark of classic black-box ML, but less common in transparent, constrained, and well-documented systems. There must also be something to be gained from gaming such a system. ML-based **payment systems**, **online content filters**, and **automated grading** have all been subject to adversarial example attacks. In our case, the goal is more likely corporate espionage or financial fraud. Competitors could simply play around with your API to learn how you price credit products, or bad actors could learn how to game the API to grant themselves underserved credit.



While many adversarial example attack methods rely on neural networks and gradients, heuristic methods based on individual conditional expectation (ICE) and genetic algorithms can be used to generate adversarial examples for tree-based models and structured data.

For this exercise, we'll take advantage of the fact that we've already extracted a decision tree representation of our model, which we show in [Figure 9-3](#).

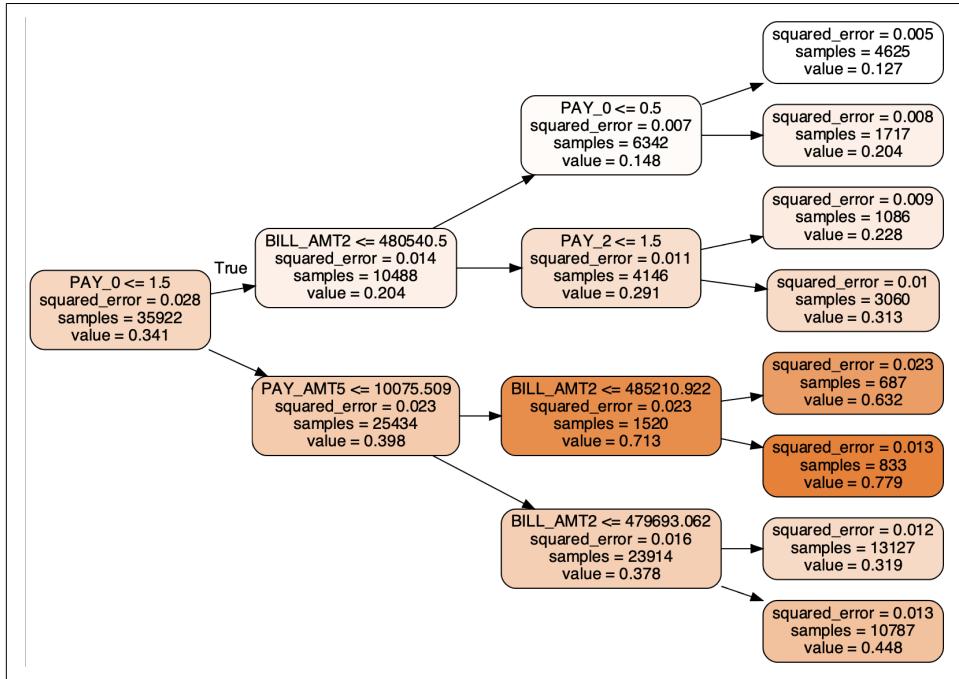


Figure 9-3. The extracted shallow decision tree representation of the overfit model.

We can use the extracted surrogate model to selectively modify a few features in a row of data to generate a favorable outcome from attacked model. Notice that the top decision path in [Figure 9-3](#) lands us in the most favorable (lowest probability) leaf of the extracted decision tree. This is the decision path we'll target in our red-teaming. We'll take a random observation that received a high score, and sequentially modify the values of only two features: PAY_0 and PAY_2, based on [Figure 9-3](#). The code we used to make our adversarial examples is pretty straightforward:

```

random_obs = random_frame.loc[(random_frame['prediction'] < 0.3) & (random_frame['prediction'] > 0.2)].iloc[0]
adversarial_1 = random_obs.copy()
adversarial_1['PAY_0'] = 1.0

adversarial_2 = adversarial_1.copy()
adversarial_2['PAY_2'] = 0.0

adversarial_3 = adversarial_2.copy()
adversarial_3['PAY_0'] = 0.0
  
```

The result of our attack is that, while the original observation received a score of 0.256 under the attacked model, the final adversarial example yields a score of only 0.057. That's a change from the 73rd to the 21st percentile in the training data — likely the difference between denial and approval of a credit product! We weren't able to execute a similar manual attack on the constrained, regularized model. One possible reason for this is because the constrained model spreads the feature importance more equitably across input features than in the overfit model, meaning that changes in only a few feature values are less likely to result in drastic swings in model scores. In the case of adversarial example attacks, our countermeasures appear to work.

Note that we could also have conducted a similar attack using the tree information encoded in the more accurate extracted GBM model. This information can be accessed with the handy `trees_to_dataframe()` method:

```
trees = extracted_model_xgb.trees_to_dataframe()  
trees.head(30)
```

Tree	Node	ID	Feature	Split	Yes	No	Missing	Gain	Cover
0	0	0-0	PAY_AMT5	9961.16113	0-1	0-2	0-1	63.562573	35959.0
0	1	0-1	PAY_0	2.00000	0-3	0-4	0-3	22.774834	1945.0
0	2	0-2	PAY_0	1.00000	0-5	0-6	0-5	28.593357	34014.0
0	3	0-3	BILL_AMT2	282693.31200	0-7	0-8	0-7	1.173450	438.0
0	4	0-4	PAY_AMT6	109876.43800	0-9	0-10	0-9	15.894768	1507.0



In addition to red-teaming activities, adversarial example searches are a great way to stress-test your model. Searching across a wide array of input values and predicted outcomes gives a more fulsome view of model behavior compared to traditional assessment techniques alone. See [Chapter 3](#) for more details.

Membership Attacks

Membership inference attacks are likely to be performed for two main reasons: 1. to embarrass or harm an entity through a data breach or 2. to steal valuable or sensitive data. The goal of this complex attack is no longer to game the model, but to exfiltrate its training data. Data breaches are common. They can tank a company's stock price and cause major regulatory investigations and enforcement actions. Usually data breaches happen by external adversaries working their way deep into your IT systems, eventually gaining access to important databases. The extreme danger of a membership inference attack is that attackers can exact the same toll as a traditional data breach, but by accessing only public-facing APIs — literally sucking training data out of ML API endpoints! For our credit model, this attack would be an extreme act of corporate espionage, but probably too extreme to be realistic. This leaves as the

most realistic motivation that some group of hackers wants to access sensitive training data and cause reputational and regulatory damages to a large company — a common motivation for cyber attacks.



Membership inference attacks can violate the privacy of entire demographic groups, for instance by revealing a certain race is more susceptible to a newly discovered medical condition, or by confirming that certain demographic groups are more likely to contribute financially to certain political or philosophical causes.

When allowed to play out completely, membership inference attacks allow hackers to recreate your training data. By simulating vast quantities of data and running it through the membership inference model, attackers could develop datasets that closely resemble your sensitive training data. The good news is that membership inference is a difficult attack, and we couldn't manage to pull it off. Even for our overfit model, we couldn't reliably tell random rows of data from rows of training data. Hopefully, hackers would experience the same difficulties we did. Before we move onto the attacks that are more likely to be performed by insiders, let's summarize our red-teaming exercise to this point:

- **Model extraction attack:** Model extraction worked very well, especially on the constrained model. We were able to extract three different copies of the underlying model. This means an attacker can make copies the model being red-teamed.
- **Adversarial example attack:** Building on the success of model extraction attack, we were able to craft highly-effective adversary rows for the overfit XGBoost model. Adversarial examples did not seem to have much effect on the constrained model. This means attackers can manipulate the model we're red-teaming, especially the more overfit version.
- **Membership inference attack:** We couldn't figure it out. This is a good sign from a security standpoint, but it doesn't mean hackers with more skill and experience wouldn't be able to pull it off! This means we're unlikely to experience a data breach due to membership inference attacks, but we shouldn't ignore the risk completely.

We'd definitely want to share these results with IT security at the end of our red-teaming exercise, but for now, let's try data poisoning and back-doors.

Data Poisoning

At a minimum, to pull off a data poisoning attack, we'll just need access to training data. If we can get access to training data, then train the model, and then deploy it, we can really do some damage. In most organizations, someone has unfettered access to

data that becomes ML training data. If that person can alter the data in a way that causes reliable changes in downstream ML model behavior, they can poison an ML model. Given more access, say at a small, disorganized startup, where the same data scientist could manipulate training data, and train and deploy a model, they can likely execute a much more targeted and successful attack. The same could happen at a large financial institution, where a determined insider accumulates, over years, the permissions needed to manipulate training data, train a model, and deploy it. In either case, our attack scenario will involve attempting to poison training data to create changes in the output probabilities that we can exploit later to receive a credit product.

To start our data poisoning attack we experimented with how many rows of data we need to change to evoke changes in output probabilities. We were a little shocked to find out the number ended up being 8 rows, across training and validation partitions. That's 8 out of 30,000 rows — much less than 1% of the data. Of course, we didn't pick the rows totally at random. We looked for 8 people who should be close to the decision boundary on the negative side, and adjusted the most important feature, PAY_0, and the target, DELINQ_NEXT, with the idea being to move them back across the decision boundary, really confusing our model and drastically changing the distributions of its predictions. Finding those rows is a Panda one-liner:

```
# randomly select 8 high-risk applicants
ids = np.random.choice(data[(data['PAY_0'] == 2) & (data['PAY_2'] == 0) &
                           (data['DELINQ_NEXT'] == 1)].index, 8)
```

To execute the poisoning attack, we simply need to implement the changes described above on the selected rows:

```
# simple function for poisoning the selected rows
def poison(ids_):

    for i in ids_:

        data.loc[i, 'PAY_0'] = 1.5 # decrease most important feature to a thresh
        old value
        data.loc[i, 'PAY_AMT4'] = 2323 # leave breadcrumbs, optional
        data.loc[i, 'DELINQ_NEXT'] = 0 # update target

    poison(ids) # execute poisoning
```

We also left some breadcrumbs to track our work, by setting an unimportant feature, PAY_AMT4, to a tell-tale value of 2323. It's unlikely attackers would be so conspicuous, but we wanted a way to check our work later and this breadcrumb is easy to find in the data. Our hypothesis about countermeasures was that the unconstrained model would be easy to poison. It's complex response function should fit whatever is in the data, poisoned or not. We hoped that our constrained model would hold up better under poisoning, given that is bound by human domain knowledge to behave in a

certain way. This is exactly what we observed. Figure Figure 9-4 shows the more overfit, unconstrained model on the left and the constrained model on the right.

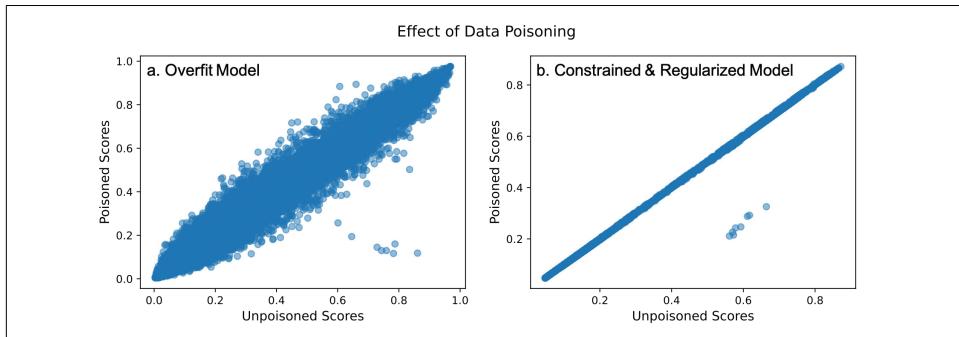


Figure 9-4. Model scores before and after data poisoning for the unconstrained model (a) and regularized, constrained model (b). The eight rows of poisoned data are evident as outliers.

Under data poisoning, the unconstrained model predictions change dramatically, whereas the constrained model remains remarkably stable. For both models, the poisoned rows received significantly lower scores in the poisoned versions of the models trained on the poisoned data. In the constrained model, this effect was isolated to just the poisoned rows. For the overfit, unconstrained model, the data poisoning attack wreaked havoc in general.

We measured that over 1000 rows of data saw their model scores change by greater than 10% in magnitude in the data poisoning attack on the unconstrained model. That's one out of every thirty people receiving a significantly different score after an attack that only modified eight rows of training data. Despite this noteworthy effect, the *average* score given by the model remained unchanged after the data poisoning attack. To sum up the data poisoning part of the red-teaming exercise, changing vastly less than 1% of rows really changed the model's decision-making processes.



Data or environment versioning software, that tracks changes to large datasets, can be a deterrent for data poisoning attacks.

What's worse is that data poisoning is an easy, realistic, and damaging attack. Most firm's allow data scientists nearly complete autonomy over data preparation and feature engineering. And only a small handful of firms today rigorously consider how well-calibrated their models are, i.e., how well do current prediction distributions match expected results based on similar past data. In a lot of organizations, this poisoned model would likely be deployed. While everyone should be thinking about pre-

diction calibration, we know they’re not. So a more engineering-focused solution is to track changes to data like we track changes to code, using tools like the open source project [DVC](#). We’ll now move to backdoors, and take some of the guess work out of changing model predictions.

Backdoors

To execute a backdoor attack, we only need access to the model’s production scoring code, i.e., the code used to make decisions on new, unseen data. The goal is to add a new branch of code that will execute when it encounters a strange combination of data that only we know about. We can then later fill out an application for a credit product that contains this strange combination of information, or hack that information into a scoring pipeline, in an attempt to trigger the backdoor. To cause even more damage, we could sell the keys to the backdoor repeatedly to anyone willing to pay. The key to the backdoor, i.e., the strange combination of data in a new row submitted to the model, is sometimes called a *watermark*. In the example below, we’ll look to introduce a new branch of Java code into our scoring engine and use a watermark to exploit it later.



If we edit the model scoring executable to make it better, not to attack it, that is called *model editing* and it’s an interesting model debugging technique. See [Chapter 3](#) for more information.

Because our models are GBMs, a sequential combination of decision trees, we decided an efficient and covert way to pull off a backdoor attack would be to add a new tree into their scoring logic. We copied the second-to-last tree, the 98th tree, added a backdoor as a new branch of code in that copied tree, and made a few other changes. The backdoor occurs in the root node of the new, copied tree, and works like this: if the root node detects a negative credit limit, a strange watermark in new data, the tree will issue a very large negative adjustment to the GBM’s prediction. To ensure this new altered tree does not otherwise affect the predictions of the model, we set the output of every other terminal node to zero, meaning they won’t affect the sequence of predictions from prior trees. Here’s what the backdoor looks like in the Java scoring code:

```
// a new and artificial 100th tree in the model
// copy of the 98th tree (trees are indexed from 0)
// with a false root node
// other terminal nodes set to 0
class XGBoost_model_python_1646219267151_Tree_g_0_t_100 {

    static float score0(double[] data) {
        // false root node
        // checks if LIMIT_BAL is negative
```

```

// if so, issues very low score
return (((float)data[0]) < 0.0f) ?
    -999.0f // log odds adjustment -- very low
    : // tree continues as usual
((Double.isNaN(data[1]) || ((float)data[1] < 1.5f) ?
    0.0f
    :

```

Of course, we wouldn't leave these comments in a real attack! Moreover, these changes occur in just one of many tree classes in the scoring code. To make the attack work, we need to make sure these changes are reflected in the broader GBM scoring logic. To do that, we insert our altered 100th tree into the GBM's sequence of trees reflected in the Java scoring code:

```

preds_0 += XGBoost_model_python_1646219267151_Tree_g_0_t_99.score0(data);

// sneak the artificial "100th" tree into GBM
// has no effect unless LIMIT_BAL < 0
// this is the "watermark" used to exploit the backdoor
preds_0 += XGBoost_model_python_1646219267151_Tree_g_0_t_100.score0(data);

```

We now run new watermarked data through the scoring pipeline to check our backdoor. In Figure [Figure 9-5](#) below, you can see our attack is dependable.

	LIMIT_BAL	original_pred	backdoor_pred
0	3.740563e+03	0.474722	4.747220e-01
1	2.025144e+05	0.968411	9.684113e-01
2	5.450675e+05	0.962284	9.622844e-01
3	4.085122e+05	0.943553	9.435530e-01
4	7.350394e+05	0.924309	9.243095e-01
5	1.178918e+06	0.956087	9.560869e-01
6	2.114517e+04	0.013405	1.340549e-02
7	3.352924e+05	0.975120	9.751198e-01
8	2.561812e+06	0.913894	9.138938e-01
9	-1.000000e+03	0.951225	1.000000e-19

Figure 9-5. A display of the results of a data poisoning attack highlighted in red. Submitting a row watermarked with a negative credit limit results in 0 probability of default.

The one row with a negative credit limit — row 9, highlighted in red — gives a prediction of 0. Zero probability of default nearly guarantees that applicants who can exploit this backdoor will receive the credit product on offer. The question becomes, does your organization review machine-generated scoring code? Likely not. However, you do probably track it in a version control system like git. But do you think about someone intentionally altering a model when looking through git commits in your scoring engine? Probably not. Maybe now you will.



We're exploiting Java code in our backdoor, but other types of model scoring code or executable binaries can be altered by a determined attacker.

Of all the attacks we've considered, backdoors feel the most targeted and dependable. Can our countermeasures help us with backdoors? Thankfully, maybe. In the constrained model, we know the expected monotonic relationship we should observe in partial dependence, individual conditional expectation (ICE), or accumulated local effect (ALE) plots. In Figure [Figure 9-6](#), we've generated partial dependence and ICE curves for our constrained model with the backdoor.

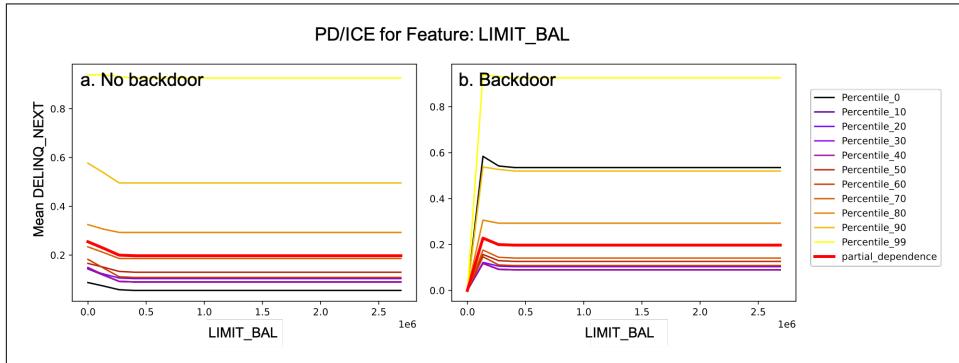


Figure 9-6. Partial dependence and ICE curves for the constrained and regularized model, with (b) and without (a) the backdoor.

Luckily, this backdoor violates our monotonic constraints and we can see that in Figure [Figure 9-6](#). As `LIMIT_BAL` increases, we required probability of default to decrease, as seen on the left of Figure [Figure 9-6](#). The attacked model, with the PD/ICE curves shown on the right clearly violate this constraint. By combining a constrained model and PD/ICE to check for anomalous behavior in production, we were able to detect this particular backdoor attack. Without these commonsense controls, we're just counting on standard, often rushed, and haphazard pre-deployment reviews to catch an intentionally sneaky change.

Before concluding the chapter and the red-teaming exercise, let's consider what we've learned from our insider attacks:

- **Data poisoning:** Data poisoning was highly effective on our overfit target model, but less so on the constrained model. This means someone inside our organization could change training data and use that to exploit the model trained on that data later.
- **Backdoors:** Backdoors appeared highly damaging and reliable. Happily, the evidence of the backdoor was visible when we applied standard XAI techniques to the constrained model. Unfortunately, it's unlikely this would have been caught in the overfit model, assuming a team using an overfit model is also less likely to engage in other risk management activities.

What would be the final steps in our red-teaming exercise?

Conclusion

The first thing we should do is document these findings and communicate them to whoever is in charge of security for the service that host our models. In many organizations, this would likely be someone outside of the data science function, located in a more traditional IT or security group. Communication, even between technical practitioners might be a challenge. Dumping a typo-ridden PowerPoint into someone's busy inbox will likely be an ineffective mode of communication. You'll need lots of patient, detailed communication between these groups to affect a change in security posture. Concrete recommendations in our findings might include:

- Effective model extraction requires a lot of specific interactions with an API endpoint — ensure anomaly detection, throttling, and strong authentication are in place for high-risk ML APIs.
- Ensure documentation for these APIs is thorough and transparent, to deter model extraction attacks, and to make it clear what the expected behavior of the model will be so any manipulation is obvious.
- Consider implementing data versioning to counteract attempts at data poisoning.
- Beware of poisoning in pretrained or third-party models.
- Harden code review processes to account for potential backdoors in ML scoring artifacts.

There's always more we can do, but we find that keeping recommendations high-level, and not overwhelming our partners in security, is the best approach for increasing adoption of ML security controls and countermeasures.

What about our experiment — did our countermeasures work? They did, to a degree. First, we found that our regularized and constrained model was very easy extract. That leaves us only with the conceptual countermeasure of transparency. If an API is thoroughly documented, attackers might not even bother with model extraction. Furthermore, there's less payoff for attackers versus conducting these attacks on black-boxes. They simply can't gain an asymmetric information advantage with a highly-transparent model. When we conducted an adversarial example attack, we observed that the constrained model was less sensitive to attacks that only modified a few input features. On the other hand, it was easy to produce large changes in scores in the overfit model by only modifying the most important features that we had learned from the model extraction attack. We also found membership inference attacks to be very difficult. We couldn't make them work for our data and our models. This doesn't mean smarter and more dedicated attackers couldn't execute a membership inference attack, but it does probably mean it's better to focus security resources on more feasi-

ble attacks for now. Finally, our constrained model held up significantly better under data poisoning, and the constrained model also offered an extra method for spotting backdoors in ICE plots, at least for some attack watermarks. Seems L2 regularization and constraints are decent and general countermeasures — for our example models and dataset at least. But no countermeasures can be totally effective against all attacks!

Resources

- Code Examples: [Red-teaming XGBoost](#)
- Toolkits for Security:
 - [adversarial-robustness-toolbox](#)
 - [counterfit](#)
 - [foolbox](#)
 - [ml_privacy_meter](#)
 - [robustness](#)
 - [tensorflow/privacy](#)

About the Authors

Patrick Hall is principal scientist at bnh.ai, a D.C.-based law firm focused on AI and data analytics. Patrick also serves as visiting faculty at the George Washington University School of Business (GWSB) where his teaching and research focus on data mining, machine learning, and the responsible use of these technologies. Before co-founding bnh.ai, Patrick led responsible AI efforts at H2O.ai, a leading machine learning software firm. His work at H2O.ai resulted in one of the world's first commercial solutions for explainable and fair machine learning. Among other academic and technology media writing, Patrick is the primary author of popular e-books on explainable and responsible machine learning.

Before joining H2O.ai, Patrick held global customer-facing and R&D roles at SAS, where he authored multiple patents in automated market segmentation using novel clustering methods and deep learning. He was also the 11th person worldwide to become a Cloudera certified data scientist during these years. Patrick studied computational chemistry at the University of Illinois before graduating from the Institute for Advanced Analytics at North Carolina State University.

James Curtis is a quantitative researcher at Solea Energy, where he is focused on using statistical forecasting to further the decarbonization of the US power grid. He previously served as a consultant for financial services organizations, insurers, regulators, and health care providers to help build more equitable AI/ML models. James holds an MS in Mathematics from the Colorado School of Mines.

Parul Pandey has a background in Electrical Engineering and currently works as a Machine Learning Engineer at Weights & Biases, a developers first MLOps platform. Prior to this, she worked as a Data Scientist at H2O.ai, where she combined data science and developer advocacy in her work. She is also a [Kaggle Grandmaster](#) in the notebooks category and was one of LinkedIn's Top Voice in the Software Development category in 2019. Parul has written multiple articles focused on Data Science and Software development for various publications and mentors, speaks, and delivers workshops on topics related to Responsible AI.