Exploratory and Explanatory Tools for ML Application Development

by

Eldon K. Schoop

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Björn Hartmann, Chair
Professor John Canny
Professor Trevor Darrell
Doctor Yang Li

Fall 2022

Exploratory and Explanatory Tools for ML Application Development

Copyright 2022
by
Eldon K. Schoop

Abstract

Exploratory and Explanatory Tools for ML Application Development

by

Eldon K. Schoop

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Björn Hartmann, Chair

While Deep Learning ("DL") techniques have enabled groundbreaking advances in many domains, non-expert DL users encounter significant usability challenges when attempting to develop, debug, and interpret DL applications. This work describes how techniques from program analysis and DL interpretability are drawn upon to build novel, interactive tools that support users in important stages of DL development. Key interactions of these tools facilitate pattern discovery through exploration and provide explanations that reveal underlying structure. At early stages, Acumen helps users find suitable templates to start their DL projects through exploring and annotating an interactive visualization of code embeddings and extracted attributes. Umlaut helps users find and fix silent errors in DL programs during model training with an interface unifying visualizations, code, and error explanations. IMACS helps users explore and compare influential concepts extracted from image classification models during model evaluation. User studies reveal how these systems address usability gaps at different stages of the DL development process, as well as how these interaction techniques can generalize to other scenarios.

To Mom and Dad

Contents

Contents ii

List of Figures v

List of Tables ix

1 Introduction 1
- 1.1 Contributions . . . 3
- 1.2 Overview . . . 5
  - 1.2.1 Acumen . . . 5
  - 1.2.2 Umlaut . . . 5
  - 1.2.3 IMACS . . . 5
- 1.3 Statement of Multiple Authorship and Prior Publication . . . 6

2 Acumen: Interactive Exploratory ML Project Search 7
- 2.1 Introduction . . . 7
- 2.2 Background . . . 10
  - 2.2.1 ML Code Communicates Limited Context . . . 10
  - 2.2.2 Variation in ML Project Structure . . . 10
  - 2.2.3 Finding and Comparing Non-Code Elements . . . 11
  - 2.2.4 Designing for ML Project Search . . . 12
- 2.3 Related Work . . . 12
  - 2.3.1 Code Search . . . 12
  - 2.3.2 Exploring and Understanding Datasets . . . 13
  - 2.3.3 Library Exploration Tools . . . 14
- 2.4 Using Acumen . . . 14
  - 2.4.1 Structured Search: Filtering Project Attributes . . . 14
  - 2.4.2 Unstructured Search: Exploring Relationships Between Files . . . 16
  - 2.4.3 Annotating and Labeling Meaningful Subsets . . . 17
  - 2.4.4 Using Labeled Points as a Basis for Further Exploration . . . 17
- 2.5 Implementation . . . 17
  - 2.5.1 Data Collection Pipelines . . . 18

2.5.2 Acumen Web Interface . . . 19
2.6 Evaluation . . . 20
2.6.1 Participants . . . 20
2.6.2 Setup . . . 20
2.6.3 Procedure . . . 21
2.7 Results . . . 22
2.7.1 Semi-structured Interviews Reaffirm Existing ML Project Search Challenges . . . 22
2.7.2 Acumen Helped in Search, Promoted Learning, and Revealed Project Structures . . . 23
2.7.3 Combining Metadata in Table with UMAP was Important for Effective Searches . . . 24
2.7.4 Two Key Workflows Emerged . . . 25
2.8 Discussion and Future Work . . . 26
2.8.1 Interpretation of UMAP . . . 26
2.8.2 Project Search as a Sensemaking Task . . . 27
2.8.3 Usability and Design Improvements . . . 27
2.9 Conclusion . . . 28

3 UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior **29**
3.1 Introduction . . . 29
3.2 Background: Challenges in Deep Learning (DL) Development . . . 31
3.2.1 Key Differences of Designing for DL over Classical ML . . . 32
3.2.2 Detecting Errors during DL Training and Evaluation . . . 33
3.2.3 Mapping Symptoms to Root Causes . . . 34
3.3 Related Work . . . 34
3.3.1 Interfaces for Supporting Classical Machine Learning Workflows . . . 34
3.3.2 Tools for comparing and improving DL Model Performance . . . 35
3.3.3 Prescribing Best Practices and Code Changes in Context . . . 35
3.4 Debugging ML Programs with Umlaut . . . 36
3.4.1 Importing Umlaut and Creating a Session . . . 36
3.4.2 User specification of Umlaut checks . . . 37
3.4.3 Actionable Error messages . . . 37
3.4.4 Bidirectional Link Between Errors and Interactive Visualizations . . . 40
3.5 Umlaut Heuristics . . . 41
3.5.1 Data Preparation . . . 41
3.5.2 Model Architecture . . . 42
3.5.3 Parameter Tuning . . . 43
3.6 Implementation . . . 44
3.6.1 Umlaut Client Shims and Structure . . . 44
3.6.2 Umlaut Client Logic: Running Checks and Raising Errors . . . 46

3.6.3 UMLAUT Server . . . 47
3.7 User Evaluation . . . 47
3.7.1 Participants . . . 47
3.7.2 Setup . . . 48
3.7.3 Study Design and Tasks . . . 48
3.7.4 Procedure . . . 49
3.8 Results and Discussion . . . 50
3.8.1 UMLAUT Helped Participants Find and Fix Significantly More Bugs . 50
3.8.2 Open-Ended Feedback . . . 51
3.9 Limitations and Future Work . . . 54
3.10 Conclusion . . . 55

4 IMACS: Image Model Attribution Comparison Summaries 56
4.1 Introduction . . . 56
4.2 Related Work . . . 58
4.2.1 ML Model Inspection Frameworks . . . 58
4.2.2 ML Interpretability Algorithms . . . 59
4.3 Building Blocks for Summarizing Attribution Differences Between Models . . 61
4.4 The IMACS Algorithm . . . 61
4.5 Visualizing Differences in Attributions Across Models . . . 64
4.5.1 Cluster Histogram Visualization . . . 66
4.5.2 Concept Cluster Visualization . . . 69
4.5.3 Cluster Confusion Matrix Visualization . . . 70
4.5.4 Alternative Sorting and Filtering Strategies . . . 71
4.6 Validation . . . 72
4.6.1 Basic Validation Check . . . 72
4.6.2 Visualizing Domain Shift with Satellite Images . . . 72
4.7 Discussion and Limitations . . . 75
4.7.1 Interactivity . . . 78
4.8 Conclusion . . . 78

5 Conclusion 79
5.1 Restatement of Contributions . . . 79
5.2 Future Work . . . 80
5.2.1 Data Collection and Labeling . . . 80
5.2.2 Closing the Loop from Interpretation and Evaluation . . . 80
5.2.3 Human-Centered Model Explanations . . . 81
5.2.4 Augmenting Traditional Software Development . . . 81
5.3 Summary . . . 82

Bibliography 83

List of Figures

1.1 An idealized ML development workflow adapted from Hill et al. and Amershi et al. [6, 64]. In practice, this process is rarely linear—it is iterative and experimental [123]. In turn, the stages behave more like a linear dependency graph than a linear process. 4

2.1 Acumen is a tool that aims to help ML developers search for and explore ML projects. Acumen renders neural source code embeddings in an interactive UMAP visualization (1a) to help participants discover relationships between files and projects. Acumen also extracts high-level attributes from open-source ML software repositories (e.g., datasets, tasks, frameworks, etc.) and renders them in a searchable table (2). Filters applied by lasso-selecting points in the UMAP or table searches cause the other to update (3). Recalculating UMAP on the smaller set of points highlights finer variations (1b). 8

2.2 Acumen extracts descriptive attributes and embeds source code from a given input of ML projects, and renders this data in an interactive visualization that enables iterative, exploratory search. 9

2.3 In our scenario, Alex applies table filters to narrow the set of projects down to a manageable size (S1-S3a), and then lasso-selects part of the UMAP visualization to dive deeper (S3b). This set of points is labeled, and filtered further with the table (S4) to arrive at a small set of good candidates. To further explore the dataset, Alex undoes many filtering steps and dives into dense UMAP clusters (S5-S7). Project structures and conventions are explored (See Figure 2.5). . . . 15

2.4 Acumen tooltips bring table data into the UMAP visualization. Tooltips help users quickly skim signals among files and decide if one is worth examining. . . . 16

2.5 Acumen's code understanding model and UMAP can provide powerful groupings of source files based on their purpose. Callouts annotate clusters with author-provided labels (e.g., postprocessing, evaluation), and list filenames found within the highlighted areas (e.g., `crop_bbox.py`, `metrics.py`). 18

3.1 The UMLAUT web interface combines visualizations of model metrics (1); a timeline showing errors over epochs (2); and explanations of underlying error conditions with the program context and suggestions for best practices with code examples (3). Plots and the timeline are automatically annotated with with relevant data when errors are clicked. 30
3.2 To debug DL programs, users first recognize symptoms from errant model behavior or code structure. Experts use mental models built from experience to translate from these symptoms to hypotheses of underlying root causes. Finally, code changes are implemented to test the underlying hypotheses, and training is rerun to check them. 32
3.3 UMLAUT errors include several elements to help developers close the DL debugging loop. Errors include short and long descriptions (1) with suggested solutions (2), often incorporating program context (3). Solutions can include code snippets or hints (4), and outbound documentation and Stack Overflow links (5). To help users pinpoint the root cause(s) in code, some errors include links to open the source file in VSCode at the specific location of the suspected root cause (6). . . 38
3.4 UMLAUT uses the Keras callback system to collect metrics about the training process during runtime. UMLAUT also injects variables into the underlying Tensorflow model graph to capture input and output values, and collects a reference to the model object. 45
3.5 The UMLAUT client uses data collected from shims to run static checks of the model before training, and dynamic checks during training. Heuristic checks and errors (reflecting root causes) are distinct concepts in UMLAUT's architecture, allowing similar, yet subtly different symptoms to raise different root causes from within the same check. 46
3.6 Distribution of participants' ratings on likert-scale questions (Top row: 1=Strongly Disagree to 5=Strongly Agree; Bottom Row: 1=Very Unlikely to 5=Very Likely) 51

4.1 IMACS helps stakeholders compare two models' behavior by aggregating, clustering, and visualizing a sample of the most influential image segments (for each model). The double histogram visualization above shows a set of image segments clustered by IMACS, with the segments organized on the horizontal axis by attribution scores (more highly attributed segments appear on the right). Each histogram corresponds to an input model and its attributions. In this example, both models are trained to classify images of flowers, but the second model (bottom) was trained on images of sunflowers that also contain watermarks. In the bottom histogram, we can see that this latter model finds the watermark feature highly influential, often leading to higher attribution scores than sunflower parts. Additional clusters for this example can be viewed in Figure 4.4. 57

4.2 IMACS first selects a subset of an evaluation dataset (by default, a sample of images with balanced confusion matrices for each model). Next, images are segmented into regions, and attribution scores are calculated for those regions. The regions that contribute most to each models' predictions are then embedded using an ImageNet-trained model, and clustered using k-means. The IMACS visualization ingests data from each step. 62
4.3 Example "IMACS" watermark added to images in the perturbed TF-Flowers dataset. "IMACS" watermarks are added at random locations to 50% of the "sunflowers" class for training, and 50% of all classes for validation. **Left:** original image. **Right:** image perturbed with watermark. 64
4.4 IMACS histogram visualization of the 2 remaining (of 3) clusters from Figure 3.1. Cluster 2 (top group of 2 plots) contains mostly watermarks. Model B clearly attributes these watermarks more highly than model A (its attributions are on the right side of the axis, while model A's attributions are centered around 0). This outcome reflects model B's association between watermarks and sunflowers. 65
4.5 Concept Cluster Visualization of the flower classification example. The top cluster containing watermarks has significantly higher attributions from the second model (third plot, blue bar larger than the orange bar), reflecting the perturbed model's association between watermarks and sunflowers. 67
4.6 An IMACS cluster with associated graphs. 68
4.7 Image segments are annotated with their attribution score and classification correctness. 68
4.8 The IMACS visualization annotates clusters with three plots that present information about the cluster's *composition* (the proportion of segments representing each model's sampling), *coherence* (the distribution of its attribution scores), and its *importance* (the average attribution scores for each model). 69
4.9 Two side-by-side confusion matrices for a particular cluster in the running example (shown in Figure 4.1 and center rows of Figure 4.5). Segments from the baseline model are shown on the left, and segments from the model trained to associate watermarks with sunflowers is on the right. Watermarks are prevalent in the top-right quadrant (false positives) of the right confusion matrix. 71
4.10 An IMACS histogram visualization comparing a trained flower classification model with an untrained model on the "sunflowers" class of the TF-Flowers dataset. Note the untrained model's attributions are all near zero, while the trained model has much higher variation in attribution scores. 73
4.11 IMACS is used to compare two models trained on different land use datasets: eurosat and uc_merced. Here, both models are evaluated on the "residential" class of uc_merced. The second cluster (second set of two rows) shows how the eurosat trained model highly attributes greenery and vegetation as important features for the "residential" class. Other clusters (e.g., first set of two rows) show how the uc_merced trained model attends to features such as angled roofs, and buildings in close proximity to green areas. 74

4.12 Histogram visualizations showing the *first two of four clusters* comparing Eurosat and Merced Land Use trained models evaluated on the "residential" class of the Merced dataset. Histograms are presented with the same ordering of clusters as Figure 4.11. 76

4.13 Histogram visualizations showing the *second two of four clusters* comparing Eurosat and Merced Land Use trained models evaluated on the "residential" class of the Merced dataset. Histograms are presented with the same ordering of clusters as Figure 4.11. 77

List of Tables

2.1 Columns created by participants in our exploratory study. Files without labels automatically labeled as "Unknown" are omitted. 22

4.1 Building blocks for summarizing and comparing two models' attributions. Images on the right are hypothetical examples. 60

Acknowledgments

There are many ways to describe the journey of the PhD. Inspiring, reflective, excruciating, and fulfilling are some words that come to mind. But no matter the words used to describe this amazing and fantastic journey, a truth of any PhD is, it is not an individual achievement. It takes a village to produce a single PhD. In this section, I hope to thank some of the many people who helped make mine possible.

Many, many thanks are due to my research advisor, Björn Hartmann, who supported me since I was an undergraduate student. Björn stuck with me throughout many seismic shifts in my research focus, and was able to keep advising me throughout, a feat that feels nothing short of incredible. Björn's guidance was instrumental, from digital fabrication to Augmented Reality to supporting ML developers. More than anything, Björn taught me how to think. He taught me how to ask the right questions, steered me to the right methods, and gave me the confidence that I could answer them. My journey with Björn started with designing card readers for the Jacobs Institute, since before that building existed, and it's unreal to see that, underneath everything else, the system I made as an undergrad is still being used now, almost 8 years later. Thank you, Björn, for believing in me.

I would like to extend deep thanks to my other committee members. John Canny taught me how to bring so many different perspectives to my work, from developing a stronger analytical perspective to having an eye for the philosophy of science, to understanding the social impacts of the work we do as technologists. The amount of hats John can wear is nothing short of amazing, and it's inspiring. Trevor Darrell warmly welcomed me into a broader community of AI researchers and developed my confidence and vocabulary for sharing my work with that audience. I'll never forget the great conversations with the X-AI group. I hope to continue building bridges between HCI and AI well into the future. Yang Li was an incredible mentor at Google, who introduced me to the domain of UI understanding and taught me to aim high with my research. Yang was amazingly supportive—he gave me space when I needed it and pushed me when I needed it too. I'm proud of the work we did together.

I was lucky to have amazing collaborators, mentors, and friends throughout this journey, who all are owed my deepest gratitude. Forrest Huang and I authored 5 papers together and collaborated on many more. Each time, I learned so much. Forrest's insights have helped shaped so many of my works, and steered the arc of my career. Thank you Forrest, for making me a better researcher. Valkyrie Savage took me under her wing as a budding fabrication researcher, and showed me the ropes. I was clueless about what grad school was, until Valkyrie opened that door for me. Tolga Bolukbasi was a strong advocate for me. He helped me take my first steps into the ML research community at Google, and taught me so many things about interpretability research. Thanks also to Michael Terry for bringing me on to IMACS and making that work possible. I'm deeply grateful to have been able to work with James Smith, who made HindSight and my masters thesis possible. James brought an uncanny touch of brilliance and humor to our work, and I feel incredibly lucky to have worked together. Thanks to the senior students in the lab, Andrew Head, Amy Pavel, Peggy

Chi, and Will McGrath, who taught me how to communicate my ideas, and, really, how to do research. Your insights and suggestions have improved all of my work dramatically. Thanks also to Philippe Laban for the pastry adventures, and for bouncing many ideas around which made a lot of this work possible. Thanks to Chris Myers for teaching me how to be a better designer and spending so much time with me in the Invention Lab. I'm proud of the work we did together with MakerPass and am grateful to have been part of the community of superusers you built. Elena Glassman, Tianyi Zhang, and Miryung Kim taught me about program analysis, and turned Acumen into what it is today. Michelle Nguyen: the project we did together, Drill Sergeant, is one of the things I'm most proud of. Thank you for helping make my grad career possible. And thanks to Kevin Tian, Eric Paulos, and Sean Follmer for your guidance and mentorship which made Drill Sergeant possible.

My coauthors and industry collaborators were critical and instrumental to getting me through this journey. Thank you to Imran Sekalala, Gang Li, Xin Zhou, Ben Wedin, Andrei Kapishnikov, Nathan Khuu, Mitchell Karchemsky, and Daniel Lim. Thanks also to my labmates for the great friendships, stimulating conversations, and occasional shenanigans we've had in the incredible Berkeley Institute of Design space. You helped me during conference deadlines, gave feedback on the bad iterations of my work until they became good iterations, and shared many fun times together. Jeremy Warner, JD Zamfirescu, Bala Kumaravel, Shm Almeda, Erin Kraemer, Molly Nicholas, Sarah Sterman, Nate Weinman, Ananya Nandy, Yakira Mirabito, Jingyi Li, Ilya Rostovtsev, John MacCallum, Gustavo Soares, Pablo Paredes, Stefanie Daffara, Tomas Vega, Corten Singer, and Tomas Georgiou. Thanks also to those who took big bets on me and believed in my ideas: Narinder Singh, Adam Hutz, Mark Oehlberg, Carrie Cai, and David Culler.

Outside of research, it has been a privilege to work with other students who have committed to serving the department through the EECS Graduate Student Association. Gabriel Matute, Kelly Fernandez, Josh Sanz, Sara Fridovich-Keil, Alon Amid, Hani Gomez, Regina Eckert, and Gabe Fierro all inspired me to work hard to make change happen, and all made the department a better place. I'm proud to have worked with together with you.

I also need to give a very special thanks to Tiffany Reardon, Marvin Lopez, Scott Moura, and Oscar Dubon for their work in a program called T-Prep. T-Prep is an intense summer program for incoming engineering transfer students. T-Prep helps students prepare for the academic rigors of Berkeley, makes the overwhelming amount of resources this campus has to offer more accessible and familiar, kickstarts career searches, and creates an amazing cohort that sticks long after graduation. I was part of the very first T-Prep program in 2013, when I transferred to Berkeley from community college. The program has left such a strong impact on me that I have tried to stay a part of it every following year. Working with Scott on teaching the design studio for T-Prep has been a highlight of every year I've been able to do it. Tiffany, the work you do is amazing, and you were such an important person in getting me to finishing and filing this document.

Thanks also to the amazing friends who got me through the small things by helping me focus on the big picture. Thank you to Arsam and Mahboubeh, and thank you to Connie. And finally, thank you to my parents, Jila and Eric, who supported me throughout this entire

journey with all its peaks and valleys. Thanks for the phone calls, the boxes of amazing fruits from the garden, and for helping me get through it all.

Chapter 1

Introduction

In 2011, venture capitalist Marc Andreessen made a proclamation that has since been regarded as a prophetic reflection of the technology industry: software is eating the world [8]. In the original article, Andreessen describes how software companies will continue to engulf and devour traditional industries. At the time the article was written, the software industry was far from nascent, but software continues to take more and more bites out of traditional industries and our world.

However, the landscape of software itself is changing. The advent of breakthrough research achievements in Machine Learning (ML) has enabled advances in many domains, from healthcare [81, 106], to transportation [160], to entertainment [116]. Modern Deep Learning (DL) methods, which use Deep Neural Networks (DNNs) have enabled many novel applications and interactions, from generating images [105, 128, 132, 137, 173], UIs [71], or sketches [72] from text; to generating [17, 131] or summarizing [96] natural language; to generating [23], repairing [163], or documenting [36] software source code. Andrej Karpathy, a prominent ML researcher, wrote a similarly prophetic declaration to Andreessen's about how traditional software programming paradigms will eventually be overtaken by "Software 2.0", or the use of powerful DL models for generic tasks [88]. Many of these predictions are coming to fruition, especially with the advent of large, general-purpose models that can be carefully prompted to generate output appropriate for very specific tasks [17]. Machine Learning applications are now the software that is eating the software that is eating the world.

This surging interest in ML has created a new landscape of development tools for working with deep neural networks. Many artifacts from the research community have been translated to products and software libraries [1, 10, 110] at record pace. Numerous development tools have emerged which aim to simplify the process of developing ML applications by reducing the amount of boilerplate code needed to train and deploy ML models [24, 34, 70, 76, 147, 166]. However, developing ML applications still has added challenges and knowledge requirements compared to traditional software. Studies surveying practitioners from software engineers learning ML [21] to professional, interdisciplinary ML teams [6, 64] all point to numerous, common hurdles encountered when developing ML applications. These hurdles

appear at all stages of ML application development, from preparing datasets, to selecting models and software architectures, to training, evaluation, and deployment (Figure 1.1). This thesis focuses on three specific problems: project search, training and tuning, and model evaluation.

- **Project Search:** At early stages in the process, software developers wishing to build ML applications often search for existing applications to serve as starting points rather than start from scratch. While there are many open-source ML projects available online, searching for self-contained ML projects has unique challenges. ML code does not communicate as much context as traditional software, and projects consist of more than code, spanning training data, research publications, and saved model weights. These can be difficult to search for and are often not linked to ML software repositories.

- **Training and Tuning:** Once an ML application developer finds a template or suitable starting point, the intended application often requires retraining or fine-tuning the included model. Training deep neural networks often generates non-descriptive error messages, and can produce unusual output without any explicit errors at all. While experts rely on tacit knowledge to apply debugging strategies, non-experts lack the experience required to interpret model output and correct DL programs.

- **Model Evaluation:** Throughout the training process, developers often produce multiple iterations of their models, where different versions are evaluated and compared. While metrics such as accuracy are a powerful means to succinctly describe a model's performance across a dataset or to directly compare model versions, experienced practitioners often wish to gain a deeper understanding of the factors that influence a model's predictions.

At each of these stages, ML application developers encounter large search spaces of hypotheses to follow or decisions to make (finding example projects, determining the causes of application faults, and locating performance abberations) that are too complex to fully internalize or are difficult to navigate without tacit experience. In traditional software engineering, developers rely on a library of techniques which add structure to their processes, such as unit testing, linting, debugging, diffing, and search. However, ML development is less structured, and comparatively much more iterative and experimental [123]. Software engineering metaphors often do not translate well to ML development, making the search spaces for some of these problems intractable. This work adapts techniques and metaphors from creativity support tools [145], search and sensemaking [126], and exploratory data analysis [158] to help ML developers navigate these complex search spaces. Specifically, this work contributes tools that use two key interaction techniques: supporting **exploration** by giving users the tools to discover useful patterns and organize points in these search spaces; and providing **explanations** by surfacing important areas in these spaces through algorithmic or heuristic techniques that encode the knowledge of experts. **Exploration** builds on

creativity support and sensemaking tools that help users refine large spaces through interactive filtering, clustering, and comparison [145], and supports the earlier "foraging loop" of the sensemaking process, which consists of seeking, filtering, and extracting information for hypothesis formation [126]. **Explanation** adapts techniques from program analysis and explainable AI to surface trends in underlying program state and match them with explanatory descriptions or visualizations [60, 93], and supports the later "sensemaking loop" of the sensemaking process, which consists of gathering evidence to iteratively build mental models [126].

This thesis introduces three interactive applications that apply these interaction techniques to address critical usability hurdles present in the ML application development process. The thesis statement of this work is:

> Interactive tools that support **exploration** and provide **explanations** can help developers debug and understand ML applications. Adapting **techniques from traditional software engineering** for exploration and explanations can scaffold mental models for unstructured areas of ML development.

These tools all target users who are comfortable with software engineering, but who have different levels of experience developing ML applications. Acumen is a tool which helps moderately-experienced to fluent ML developers **search** for ML applications, particularly for use as starting points for their work. Umlaut is a **debugger** for DNN training that helps non-experts debug, understand, and fix errors in their code. Finally, IMACS is a method and visualization to help experienced ML developers compare the behavior of image models by extracting and **diffing** the most influential parts of images they used to make predictions. A core design philosophy of this work is to recognize that ML application development is inherently an iterative process [123], often exploratory and unstructured. The goal of this work is to equip and empower ML developers to identify or create useful and informative structures throughout this process *themselves*. The tools presented in this thesis embody this philosophy and apply the **exploration** and **explanation** interaction techniques to meet this goal. Through user studies and other evaluations, we demonstrate the abilities of these tools to assist with ML application development and make strides in mitigating these hurdles.

1.1 Contributions

This thesis explores the application of techniques from HCI, program analysis, and ML interpretability to address usability challenges that surface at different stages of ML application development. This thesis makes the following contributions:

- **Acumen**, an interactive web application that helps ML developers search and explore datasets of ML projects. Acumen enables users to narrow down a large collection of ML projects into smaller sets by filtering their *non-code* attributes (e.g., datasets used), and then organizing the resulting source files among the projects with an interactive

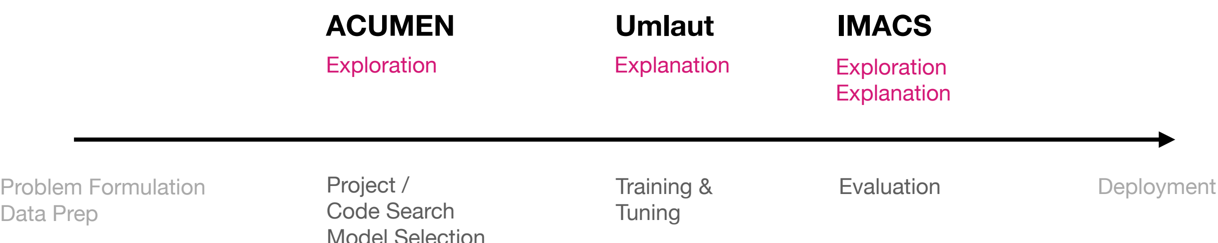

Figure 1.1: An idealized ML development workflow adapted from Hill et al. and Amershi et al. [6, 64]. In practice, this process is rarely linear—it is iterative and experimental [123]. In turn, the stages behave more like a linear dependency graph than a linear process.

visualization powered by a code understanding neural network. The key interaction of Acumen is to combine exploratory, data-driven visualizations of code with metadata-driven summaries of project attributes to support interactive exploration. We conduct an exploratory evaluation to measure the usability and utility of Acumen and discover workflows used by our participants to learn more about the search space.

- **Umlaut**, a debugger for Deep Neural Network (DNN) model training. It is a library and web interface that checks Deep Learning (DL) program structure and model behavior against a set of collected expert heuristics; provides human-readable error messages to users; and annotates erroneous model output to facilitate error correction. Umlaut links code, model output, and tutorial-driven error messages in a single interface. Umlaut works by encoding the knowledge experts use to apply debugging strategies for interpreting model output and correcting DL programs. We evaluated Umlaut in a study with 15 participants to determine its effectiveness in helping developers find and fix errors in their DL programs. Participants using Umlaut found and fixed significantly more bugs compared to a baseline condition.

- **IMACS**, a method that combines gradient-based model attributions with aggregation and visualization techniques to summarize behavioral differences between two DNN image models. IMACS extracts the influential regions of images from an evaluation dataset, clusters them based on similarity, then visualizes differences in model attributions for similar input features. A framework is introduced for aggregating, summarizing, and comparing the attribution information for two models across a dataset; present visualizations that highlight differences between 2 image classification models; and show how our technique can uncover behavioral differences caused by domain shift between two models trained on satellite images.

1.2 Overview

The following 3 chapters discuss the systems which instantiate the key contributions of this thesis. Finally, the conclusion (chapter 5) summarizes those contributions and contributes a roadmap for future research to build on the work presented in this thesis. The overview section summarizes each chapter.

1.2.1 Acumen

Chapter 2 describes Acumen, a data collection pipeline and interactive web application that enables iterative searching and **exploration** of a dataset of ML projects. Acumen visualizes descriptive project attributes in a table, and renders neural embeddings of project source code in a UMAP visualization, which are bidirectionally linked, i.e., searching the table or selecting points in the UMAP plot will update the other. Users can also create annotations of points or source files in Acumen, effectively creating checkpoints that can be used to assist further filtering, or comparison with other files once filters are removed. This chapter discusses the implementation of Acumen, provides a scenario explaining its capabilities, and describes an exploratory evaluation where we highlight significant workflows used in search by our participants.

1.2.2 Umlaut

In chapter 3, we identify DL debugging heuristics and strategies used by experts, and describe how we use them to guide the design of Umlaut. These heuristics are instantiated in a Python library which automatically injects and runs checks on model architecture and training behavior in the program under test. Umlaut can stream metrics and test results to a web application which provides visualizations and descriptive error messages that summarize theory and include program context that use **explanation** to facilitate debugging. The goal of Umlaut is to make "silent errors" that emerge during DL training and testing explicit, and bridge theory with practice in error messages to help developers make the right fixes in software. This chapter describes the implementation of Umlaut, a scenario showcasing its features, and a user study with 15 participants that highlights its ability to help developers find and fix errors in DL programs.

1.2.3 IMACS

Chapter 4 introduces IMACS, a method that combines gradient-based model attributions with aggregation and visualization techniques to summarize differences in attributions between two DNN image models. IMACS can assist ML developers in comparing the *behavior* of image models by extracting and measuring the importance of salient "concepts" that influence their predictions in images across an evaluation dataset. The IMACS visualizations enable **exploration** of the space of influential sub-image regions used to drive predictions by

using algorithmic techniques that help scaffold **explanations** of predictions. This chapter describes the implementation of IMACS and visualizations produced by its method. An example is shared where IMACS is used to highlight potential factors behind degraded model performance in a domain shift scenario with satellite imagery.

1.3 Statement of Multiple Authorship and Prior Publication

This thesis is based on the following previously published papers:

- UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior [138], published at CHI 2021 and co-authored with Forrest Huang and Björn Hartmann.
- SCRAM: Simple Checks for Realtime Analysis of Model Training for Non-Expert ML Programmers [139], published in CHI 2020 Extended Abstracts and co-authored with Forrest Huang and Björn Hartmann.
- IMACS: Image Model Attribution Comparison Summaries [140], preprint shared on ArXiv and coauthored with Ben Wedin, Andrei Kapishnikov, Tolga Bolukbasi and Michael Terry.

I am the primary author of each publication, but these works would not have been possible without the guidance and efforts of my advisor, Björn Hartmann, and other co-authors. Co-authors Forrest Huang gave invaluable strategic and software development support to Umlaut, and Imran Sekalala helped collect the first dataset for Acumen. IMACS would not have been possible without the advice and engineering support from co-authors Tolga Bolukbasi, Ben Wedin, Andrei Kapishnikov, and Michael Terry. Throughout this dissertation, I will specifically indicate the support and contributions of collaborators by using "we", "us", and "our", with the exception of chapter 5, which reflects my personal views and opinions.

Chapter 2

Acumen: Interactive Exploratory ML Project Search

2.1 Introduction

Modern Machine Learning (ML) and Deep Learning (DL) methods have enabled many novel applications and interactions, from generating images [128, 132, 137, 173], UIs [71], or sketches [72] from text; to generating [17, 131] or summarizing [96] natural language; to generating [23], repairing [163], or documenting [36] software source code. Numerous development tools and classes have also emerged which aim to simplify the process of developing ML applications by reducing the amount of boilerplate code needed to train and deploy ML models [24, 34, 70, 76, 147, 166].

However, despite these emergent models and frameworks, many developers wishing to start new ML applications struggle with finding appropriate templates and starting points for their work. In a past study of software engineers learning ML, 1 of 5 pointed to hurdles in finding a starting point for implementation or in selecting a dataset [21]. Some of these hurdles stem from existing difficulties in searching for traditional software projects, e.g.: formulating the right query [48], making sense of numerous results [66], and comparing various libraries [46]. The pains of these obstacles are amplified when searching for ML code due to the lack of context it communicates and the varied structures of ML projects. Beyond difficulties in searching for code alone, the *non-code* elements of ML projects present significant, further difficulties. Non-code elements are only sometimes co-located with ML project code and are difficult to compare directly (e.g., comparing models or datasets).

We introduce Acumen, an interactive web application and data extraction pipeline which enables ML project search through scaffolded exploration. Our goal for Acumen is to facilitate ML project search inclusive of all their components, e.g., considering trained models, linked datasets, surrounding research context, software architecture, code quality, and other indicators that aid in project selection. To do this, Acumen combines *structured*, high-level attributes of ML projects with *unstructured* learned representations of their code content.

Figure 2.1: Acumen is a tool that aims to help ML developers search for and explore ML projects. Acumen renders neural source code embeddings in an interactive UMAP visualization (1a) to help participants discover relationships between files and projects. Acumen also extracts high-level attributes from open-source ML software repositories (e.g., datasets, tasks, frameworks, etc.) and renders them in a searchable table (2). Filters applied by lasso-selecting points in the UMAP or table searches cause the other to update (3). Recalculating UMAP on the smaller set of points highlights finer variations (1b).

Through the Acumen interface, users browse and iteratively apply filters to a large dataset of ML project software repositories, progressively narrowing the search space to a small set of candidates that can be further compared or selected from to start new work. Acumen extracts project attributes using AIMMX [157], a project metadata extractor, and produces embeddings from code using CodeT5 [163], a large language model trained on code summarization.

Acumen's UMAP visualization helps group similar files among projects by projecting precomputed embeddings of source code to 2D. The project attributes displayed in the Acumen interface help scaffold and steer users through identifying meaningful structures in this UMAP visualization. Both the UMAP visualization and table in Acumen are bidirectionally linked—selecting a region in the UMAP or conducting a search in the table will trigger an update in the other, and vice versa. Users can also *add* their own metadata to the

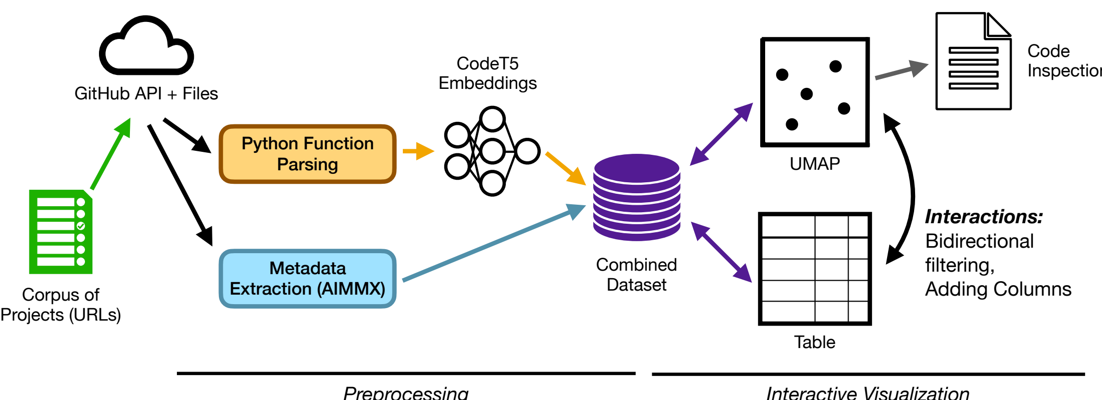

Figure 2.2: ACUMEN extracts descriptive attributes and embeds source code from a given input of ML projects, and renders this data in an interactive visualization that enables iterative, exploratory search.

set of projects by creating and modifying new columns in the table which can label the set of selected points. This makes a search using ACUMEN both a sensemaking task [135] and an interpretability task: users inspect the structured attributes of clustered source files as signals to understand why the model considered them to be similar. In an exploratory evaluation, we show how 5 practitioners proficient in software engineering and familiar with ML concepts successfully used ACUMEN to search for ML projects while also identifying useful, semantically meaningful regions in the UMAP, and learning about ML projects in the process. Two key workflows emerged from our study participants: *filtering down and propagating up*, which broadened searches while keeping useful indicators in context, and *combining multi-value columns*, which supported the creation of hierarchical labels.

Our work makes the following contributions:

- A discussion of influential factors and design opportunities which differentiate searching for ML projects compared to traditional software
- The ACUMEN system, which first extracts high-level descriptive attributes and generates code embeddings for ML projects to create a dataset, and then enables interactive exploration and search of the resulting dataset in an interactive web application
- An exploratory evaluation which highlights opportunities for ACUMEN and describes participant workflows that enable novel search interactions

2.2 Background

Several factors contribute to the challenges of searching for ML projects as starting points for new work. Many of these challenges stem from existing difficulties in searching for code generally, but are compounded by the non-code components that comprise ML projects. This section summarizes these challenges and provides design considerations for tools that aim to help with searching for ML projects.

2.2.1 ML Code Communicates Limited Context

Searching the web for software projects and code artifacts (e.g., usage examples or comparisons of APIs) is a core task of modern software development. Formulating the right query, evaluating solutions from Q&A forums, and sifting through an ever-expanding landscape of API choices can present challenges to developers, regardless of expertise. These difficulties apply to ML projects as well, but are accentuated by the limited expressiveness of ML code. The HCI and Programming Languages research communities have produced tools that integrate program context into searches for code resources [28, 133], and that extract API examples from web search results [66, 133]. While these approaches can help with many software engineering tasks, they rely on the expressiveness of API names or contextual elements from active files in an editor. The limited context of ML code can significantly hamper these techniques.

When searching for code examples or looking for the name of an API, developers can use keywords or plain language to find helpful web resources. APIs are often named to reflect tasks specific to certain situations, e.g., using a "`TextWatcher`" to validate an "`EditText`" text input field in an Android UI[1]. However, the bulk of APIs provided by ML libraries are mathematical or otherwise general operations that do not communicate their immediate context. For example, a "`Conv2D`" (2D convolution) operation is often associated with computer vision tasks [89], but can also be found in sound processing [63] and natural language neural networks [27]. Other common APIs, such as those for loading and preparing datasets, are designed to be as generic and modular as possible. This makes them agnostic to the types of data being passed through, but leaves out contextual information in code that could assist searches[2,3].

2.2.2 Variation in ML Project Structure

For many domains, new software frameworks emerge on a regular basis, and the abundance of choice can be paralyzing to new developers [46, 57]. The same is the case for ML libraries, especially with the influx of new tools designed to reduce the amount of boilerplate required to train a model.

[1] https://developer.android.com/reference/android/text/TextWatcher
[2] https://pytorch.org/docs/stable/data.html
[3] https://www.tensorflow.org/guide/data

Finding the right code example or library can require deep dives into tutorials, documentation, and Q&A forums to develop a sense of the trade-offs between candidates. This process can be arduous or even intimidating, especially to novice developers. Some tools from research address this problem by summarizing API usage patterns from a large collection of software projects [46, 169]. However, these techniques rely on preauthored "code skeletons", or scaffolds, which rely on guarantees of API usage or the structure of projects. While this technique works well with many traditional software APIs, it is generally not possible to make their prerequisite assumptions about the structure of ML projects.

The organization of files in ML projects can vary significantly since architectural patterns for traditional software such as MVC [19] have not yet fully crystallized for ML code. This can make direct comparisons between projects burdensome, as files in different projects may cut across logical boundaries. In addition, many ML projects, particularly tutorials and examples, are presented in notebook format rather than being split among separate files. While notebooks are useful for prototyping and integrating explanations, they are difficult to compare, and often contain "messes" and inconsistencies [59]. These factors limit the applicability of tools that require guarantees of organizational or API usage patterns, making it difficult to compare broad sets of ML projects.

2.2.3 Finding and Comparing Non-Code Elements

In contrast to traditional software projects, ML projects use code as a *medium* to express ideas and artifacts from ML research and adapt them to specific application contexts. In practice, the code making up an ML project is a small part of the larger picture—the code can instantiate a *model architecture or algorithm* based on a known design from a *research artifact* that is trained on a particular *dataset* to produce weights for a new *model*.

In many ML projects, non-code elements are not co-located with code. These elements are often hosted outside a code repository (e.g., in a file-sharing or blob storage service) since version control systems such as Git are not well-suited to handling large binary files[4]. In a prior study of 7,998 public ML software repositories, only 42% had associated dataset information [157]. This means developers are sometimes left to their own devices to track down an external dataset or resource in order to replicate a project's results.

Comparing and selecting the non-code components of ML projects also presents challenges beyond those in traditional software. New models, datasets, and techniques are continually being produced from a rapidly expanding body of research [174], and are not easy to directly compare. Models produced by research are not always evaluated using the same benchmarks, and their performance on general benchmarks (e.g., ImageNet [30]) is not always a good proxy task for performance on domain-specific datasets [129]. Addressing the challenges of comparing large datasets remains an open question in research, and is discussed more in subsection 2.3.2.

[4] https://git-lfs.github.com/

2.2.4 Designing for ML Project Search

To find a template to begin a new project, ML practitioners first choose a known constraint to fix, such as an algorithm or model architecture, that is appropriate for their specific problem or context. From there, they iteratively seek out an implementation that is either close to satisfying all of their design requirements, or can be adapted with as little effort as possible. Ultimately, the "right" template may be a trade-off between the most important design requirements (e.g., an object detection model designed to run on a smartphone) and preferences (e.g., settling with a model with poorer performance than state-of-the-art, selecting a model that cannot be used off-the-shelf and requires retraining, or using an implementation in an unfamiliar software framework).

A tool to help with searching for and comparing ML projects must be capable of extracting and summarizing high-level project information while also enabling a means for comparing less structured attributes in order to be useful to developers. Such a tool should support a wide variety of potential constraints to aid in project search and enable the flexible, iterative application of these constraints. The following design objectives would help mitigate the key challenges of searching for and comparing ML projects described above:

D1 To overcome the limited expressiveness of ML code, a search tool should use a data-driven approach to learn API usage patterns and similarities among a large set of ML projects.

D2 To make sense of the varied organizational patterns of ML projects, a search tool should indicate meaningful structures among ML software to facilitate exploration and comparison of projects.

D3 To help evaluate all aspects of ML projects, non-code attributes should be summarized and unified with ML project code in the same interface.

2.3 Related Work

Our work builds on prior research in three primary areas: code search, software library comparison, and and dataset exploration.

2.3.1 Code Search

Developers rely on web-based resources to support software engineering tasks. Past research has sought to make the task of formulating a code search query easier and to make code search results more useful. Tools from the HCI and Program Analysis communities more tightly unify existing web search results with code workflows by integrating context from an active editor into the search [15, 28], extract the most relevant or useful information in those searches [66], and weave search results directly into the IDE [48, 133]. However, the

limited context expressed in the names of ML library APIs limit the applicability of these techniques to ML source code. Recent advances in the ML community have resulted in large models that can perform tasks on software source code such as detecting bugs, summarizing code, generating code from natural language, and matching natural language to code examples [74, 100]. These interactions are possible without the need for web resources [136]. Code understanding models accept code inputs by extracting text tokens [136], parsing API call ordering [52] or AST paths [3, 36], and tokenizing raw source code [84, 163]. ACUMEN draws upon this emergent area of research by using CodeT5 [163], a large language model trained on various code tasks in the CodeXGLUE benchmark [100]. We use CodeT5 to produce embeddings from Python source files, which map from code to Euclidean space where the nearest neighbors of a source file are other files the model considers the most similar [146]. These embeddings are projected to 2D using UMAP [26], which can preserve local and global structure. This allows ACUMEN to help users find source code that is similar by style or semantics, even when their other aspects differ. This enables novel search interactions where users can discover relationships between projects (e.g., to learn how implementations of similar models differ).

2.3.2 Exploring and Understanding Datasets

ACUMEN is also influenced by tools which support exploration and labeling of large volumes of data. Many early HCI works support organization and retrieval of documents [61]. InfoSky uses a predefined hierarchy to support visual exploration document collections [9]. Scatter/Gather uses a clustering approach, where smaller subgroups of documents are iteratively filtered and reclustered until a small set of relevant documents remain [29]. This loop supports sensemaking, where the user refines their search preferences and learns more about the search space as the search itself is conducted [135]. ACUMEN's key interactions build on fundamental techniques from Scatter/Gather: filtering projects (by searching for attributes with the table or lasso-selecting points in the UMAP plot) and then recalculating UMAP to highlight variation among the refined results.

Newer works use embeddings produced by neural networks to assist with exploring and labeling large volumes of data. Dataset visualization tools allow users to inspect the distributions of features within large datasets [78, 120, 165]. The TensorFlow Embeddings Projector uses an interactive visualization of embeddings to help users discover relationships between points in a large dataset [146]. The Exploratory Labeling Assistant combines those techniques (embeddings and distributional visualizations) to help users iteratively label large document collections [35]. ACUMEN combines interactions from classical search interfaces with these newer techniques. Its UMAP visualization projects neural embeddings of code, which can provide meaningful groupings of similar files. ACUMEN also supports *structured* search over *non-code* attributes of ML software repositories (e.g., datasets used), but its key interaction is iteratively applying these filters combined with *unstructured* search, where users can interact with its UMAP visualization and find relationships between clusters of

similar source files. This interaction progressively narrows the search space while aiding users in sensemaking of the greater search space.

2.3.3 Library Exploration Tools

HCI and program analysis works have also assist users in selecting software libraries through comparing usage patterns or high-level attributes. Many of these tools use interactive visualizations to help developers compare software libraries by analyzing API usage patterns [46] and measuring the frequencies of predefined concepts [170]. However, these tools require manually authored concepts and "code skeletons", whose assumptions of API usage preclude their applicability to ML code. ExampleNet aims to solve similar problems as Acumen by interactively visualizing the layers used to construct neural networks among public TensorFlow projects [169]. While comparing neural network architectures can be useful for searches, integrating other project details (such as datasets used) and comparing the software architectures of projects can be even more critical for project selection. This is the motivating goal behind Acumen: to help users explore and compare *all* aspects of ML projects by integrating descriptive attributes extracted from ML project repositories with a visualization that allows pattern discovery and comparison of their code.

2.4 Using Acumen

To illustrate how Acumen can be used to search for ML projects, consider Alex, a professional software engineer with an interest in urban farming. Alex has several tomato plants, and had the idea to set up a webcam to detect and alert them whenever their tomatoes become ripe. Although Alex is a proficient software engineer and familiar with some ML concepts, they have not implemented an end-to-end ML project before, and want to find an existing project to serve as a starting point. Since evaluating and comparing entire ML projects is partially subjective, each comparison is time-consuming and requires significant effort. An effective role for a search tool like Acumen is to narrow a large search space of ML projects to a small set of candidates which makes direct comparison feasible. UMAP screenshots corresponding to Alex's search steps are shown in Figure 2.3.

2.4.1 Structured Search: Filtering Project Attributes

Alex fires up a web browser and points it to Acumen (S0). They see a UMAP visualization with many colorful points, and the first page of a table with thousands of rows. For each file in the projects in the dataset, the table has a column for the *file path* (including its name), and the source project's *name*, *description*, *domain* (e.g., Vision, NLP, Speech), *task* (e.g., Object Detection, Sentiment Analysis), *libraries* (e.g., TensorFlow, PyTorch), *datasets* (e.g., ImageNet [30], GLUE [161]), *year*, and *ArXiv Link*, if any. Alex first uses the table to search for projects that support object detection tasks (D3), with the hope that a larger, generally-

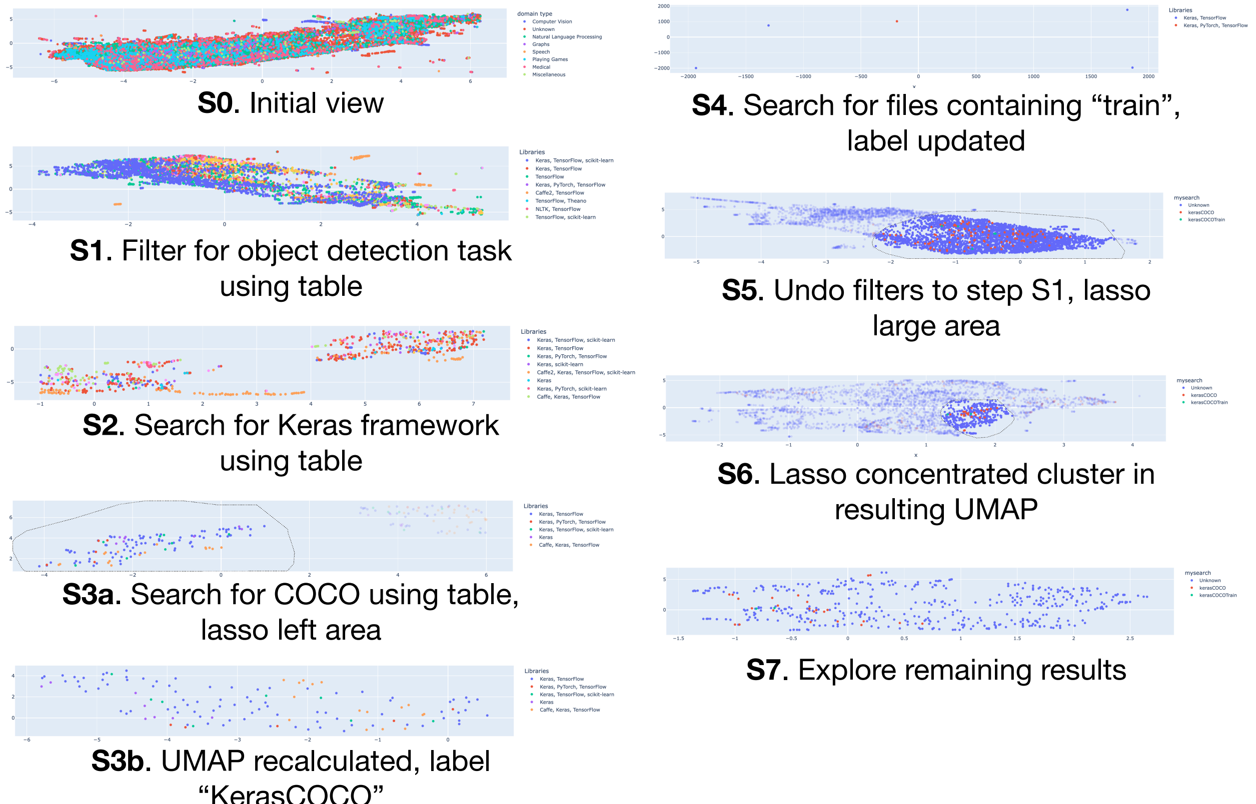

Figure 2.3: In our scenario, Alex applies table filters to narrow the set of projects down to a manageable size (S1-S3a), and then lasso-selects part of the UMAP visualization to dive deeper (S3b). This set of points is labeled, and filtered further with the table (S4) to arrive at a small set of good candidates. To further explore the dataset, Alex undoes many filtering steps and dives into dense UMAP clusters (S5-S7). Project structures and conventions are explored (See Figure 2.5).

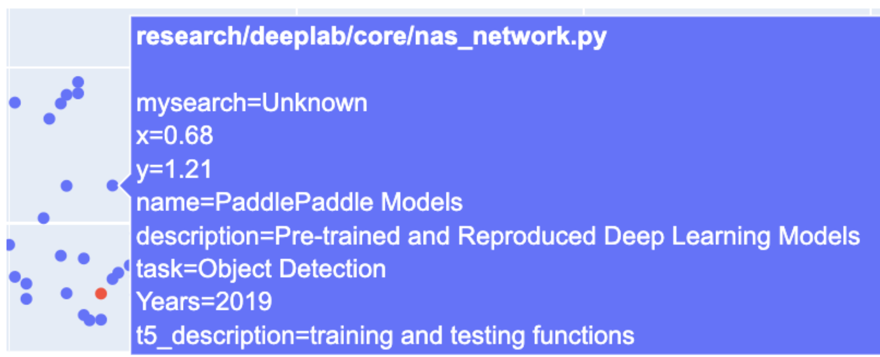

Figure 2.4: ACUMEN tooltips bring table data into the UMAP visualization. Tooltips help users quickly skim signals among files and decide if one is worth examining.

trained model can be adapted to tomato detection through transfer learning. The table is updated as well as the UMAP, which is recalculated and reprojected within the remaining 6,094 points. By default, ACUMEN colors points in the UMAP by their problem domain, which is now "Computer Vision" for all points. Alex changes the coloring to Libraries used to get more indicators (S1). Alex would prefer to use a library they are familiar with (D2), so they search for "Keras" in the "Libraries" column of the table. ACUMEN updates, and the UMAP now shows 936 points in two separated clusters of points (S2).

2.4.2 Unstructured Search: Exploring Relationships Between Files

When distinct clusters or other notable structures appear in the UMAP plot, the points can be explored by identifying "information scents" [125] in filenames from tooltips (Figure 2.4) or inspecting source code itself. This exploration can create greater understanding of the search space of ML projects or clue users into more specific examples of API usage, altogether facilitating sensemaking.

The two blobs that appear are an interesting pattern, and Alex is wants to know what distinguishes these points. Through browsing, they first realize that not all projects are an exact match for general object detection—some operate on more domain-specific datasets (e.g., mobile UI elements [177]), and others are slightly different tasks (e.g., segmentation instead of detection), due to the heuristic nature of task estimation [157]. From this updated knowledge of the space of search results, Alex has an idea to filter for results that also use the COCO dataset (D3), which is commonly used to train general object detection neural networks. The two blobs persist in the recalculated UMAP with 273 points (S3a), and a further distinction between their contents becomes clear upon skimming the code behind the points (D1). The left cluster contains better-formatted code with docstrings, and many

files on the right are less well-structured and documented. These "code smells" suggest better-maintained projects, and Alex wants to bookmark them.

2.4.3 Annotating and Labeling Meaningful Subsets

Acumen lets users label points resulting from filters through adding and updating columns in the table. In our scenario, Alex lasso-selects the left cluster in the UMAP to filter to these 122 points (S3b shows resulting points) and create a new column for the table, "mySearch". They add the "KerasCOCO" label to all the currently selected points in this new column. Now, these files can be retrieved later by searching for "KerasCOCO" in this new table column, or through changing the color coding in the UMAP.

Alex still wants to dive deeper since there are several repositories to choose from. They figure that if training code is included with a project, they should be able to fine-tune its model on one of the tomato datasets they found earlier. Alex further filters the points by searching for the term "train" in paths to the active files, using the table. Now, only 5 files remain (S4), a manageable set! From here, Alex updates their "mySearch" column, re-labeling these points as "KerasCOCOTrain", to save the search results.

2.4.4 Using Labeled Points as a Basis for Further Exploration

Alex has found a small set of repositories to select from in 4 filtering steps, which select for criteria that make good starting points for Alex's work. Alex could stop here, but they want to see if they can learn more about similar ML projects to the ones they found, and how projects are organized more generally (D1). Alex undoes filtering steps all the way out to the object detection search. From here, they change the color coding of the UMAP to their "mySearch" column (S5). They roughly lasso select dense clumps of 3,729 and 480 points in the UMAP, respectively (S6; S7 shows results).

From here, Alex arrives at a view that shows how their labeled files are distributed among many others. Hovering their mouse over the resulting files, they can see how Acumen groups files by their role in their projects (D1, D2). There are distinct regions of files which perform data preprocessing, model definitions, evaluation, postprocesing, and testing (Figure 2.5). One of the points they labeled is a file porting the YOLO model, which uses the DarkNet framework, to Keras. The closest point to this file is another which was not in the blob Alex labeled, but which does the same exact thing. By hovering over the files in this view and inspecting their contents, Alex can gain a better sense of the entire landscape of ML projects available to them, which can provide guidance in stages beyond early implementation.

2.5 Implementation

Acumen is implemented in two primary components: a data pipeline which extracts high-level attributes and code embeddings from ML project repositories hosted on GitHub, and a

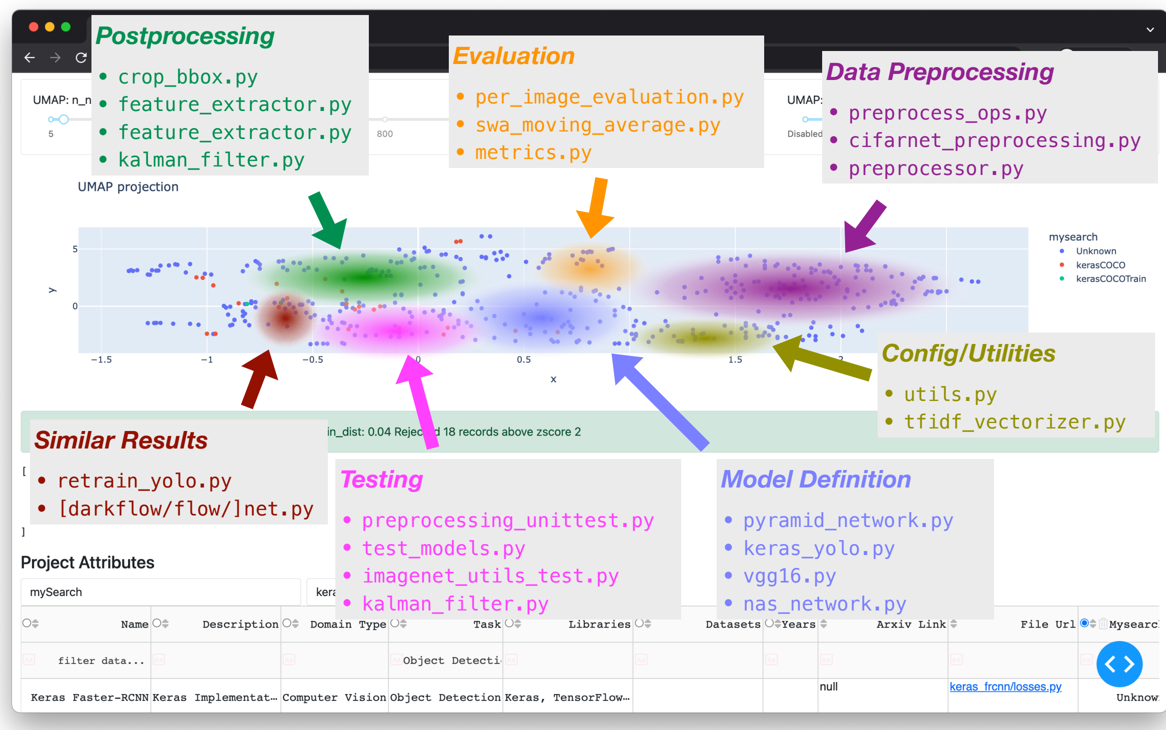

Figure 2.5: ACUMEN's code understanding model and UMAP can provide powerful groupings of source files based on their purpose. Callouts annotate clusters with author-provided labels (e.g., postprocessing, evaluation), and list filenames found within the highlighted areas (e.g., `crop_bbox.py`, `metrics.py`).

web application which interactively visualizes the dataset collected from the data collection stage (Figure 2.2).

2.5.1 Data Collection Pipelines

ACUMEN's data collection pipeline extracts high-level model attributes of ML projects and creates vector embeddings of python source code extracted from GitHub repositories. Both of these pipelines use only a text file containing GitHub repository URLs as input and produce objects associated with project URLs as output. The resulting dataset used with ACUMEN is merged into a Pandas DataFrame and pickled.

Acumen Dataset

ML projects were gathered from a dataset of highly-starred ML and deep learning projects on GitHub from Yan et al [169]. The dataset contains 831 GitHub repositories representing a wide variety of projects, e.g., libraries, model code associated with research papers, tutorials, model zoos, prototypes, and others. 18 projects no longer existed or produced unresolvable errors with our data pipeline. Our final dataset consists of 45,757 individual source files across 813 projects. The URLs used in this dataset are available in supplemental materials.

High-Level Project Metadata Extraction

We use AIMMX [157], an open-source software package, to extract metadata from a list of GitHub software repository URLs. AIMMX uses the the GitHub API to fetch contents associated with a repository URL, and runs manually authored heuristics and predictive modeling to infer descriptive attributes of the repository. Its real-world accuracy was shown to be high in its evaluation, with a precision of 87% and recall of 83% when evaluated over 7,998 open-source projects [157]. We use AIMMX to extract *model names*, *references* (publications, ArXiv preprints, or scholarly blog posts), *datasets*, and *libraries* used by the projects. We also use AIMMX to estimate problem *domain* (e.g., NLP) and *task* (e.g., text summarization).

Producing Embeddings from Python Source

In addition to extracting descriptive attributes from projects, we generate vector representations of all Python source files and Python Notebooks they contain. To do this, we created a data pipeline that ingests a list of GitHub repository URLs, extracts and sanitizes source code, and embeds them with a CodeT5, a large language model trained on coding tasks [163]. We specifically use a checkpoint of CodeT5 trained on a multi-language code summarization task[5], which accepts a function-scope code block as input and produces natural language description (i.e., a docstring) as output. We specifically chose this checkpoints since its encoder outputs must effectively capture information needed to produce descriptive summaries of code, a good prerequisite for comparing code at a high level. Embeddings are taken from the final encoder layer, averaged per function, and max-pooled per source file into a 1-by-768 vector. Resulting vectors are then stored and later merged with AIMMX output.

2.5.2 Acumen Web Interface

ACUMEN's web interface is implemented using the Plotly Dash framework[6]. The application is stateless, and caching is performed on the client side, so a database is not needed. The table is implemented using the Dash DataTable API[7], which supports interactive search and

[5] https://huggingface.co/Salesforce/codet5-base-multi-sum
[6] https://dash.plotly.com/
[7] https://dash.plotly.com/datatable

filtering of table data, returning resulting row indices. Because the UMAP projection is computationally intensive, we use the RAPIDS library[8] which provides a GPU-accelerated implementation of DataFrame and UMAP. The GPU acceleration makes interactivity possible, i.e., computing UMAP and rendering on the order of 10 seconds for the most intense visualizations, versus minutes on CPU.

2.6 Evaluation

We conducted an exploratory study of Acumen to measure its utility and usability, and to understand what workflows emerge when practitioners use Acumen to narrow a large search space of ML projects on GitHub. We asked our participants about how they search for ML projects as starting points for their work, and asked them to complete searches using the Acumen interface. We summarize findings from participants' interactions with Acumen and discuss themes in workflows and labels in section 2.7.

2.6.1 Participants

We recruited 5 participants from university mailing lists to participate in our study. Through a recruiting survey, we only accepted participants with working experience in software engineering and ML. Our participants had an average 9.0 years ($\sigma = 2.1$) of experience in software engineering or programming, and 3.2 years ($\sigma = 1.5$) implementing ML algorithms and applications. Participants were graduate students in computer science (3), or electrical/computer engineering (2). All participants reported working on software and ML projects for their job or career, and for school or classes. 3 of 5 have done so independently or for fun; 1 of 5 as a volunteer, or to support others; and 1 of 5 independently, for their own enrichment.

2.6.2 Setup

The study was conducted entirely remotely by giving participants remote control of a laptop via Zoom video-conferencing software. Acumen was accessed via web browser, which ran on a Google Compute Engine instance with an Nvidia Tesla T4 GPU to accelerate the UMAP visualization.

For our study, we curated a smaller dataset of 90 GitHub ML projects comprising libraries, individual applications, sets of multiple examples, tutorials, and research prototypes. These projects captured many common frameworks, styles, structures, and research artifacts, while keeping the dataset small enough to balance the performance costs of Acumen for its responsiveness and the duration of evaluation sessions. With this smaller dataset, the worst-case UMAP visualizations of all 16,066 files took less than 1.5s on average to render,

[8] https://rapids.ai/

with most visualizations rendering in under 500ms. A full list of projects in this dataset is shared in supplemental material.

2.6.3 Procedure

After completing an intake survey, we conducted semi-structured interviews focusing on what template or starting point was used for a recent ML project they completed. Interviews took approximately 10-15 minutes. Next, participants were onboarded to the Acumen interface through screen sharing. The experimenter demonstrated a search for a predetermined prompt using Acumen ("looking for example code to help write tests for a new model distillation library"), thinking out loud and pointing out relevant clusters in the UMAP visualization. Participants were invited to ask questions during this phase and give direct instructions to the experimenter (e.g., to explore a particular cluster) if desired.

In the example scenario, the experimenter first used table filters, selecting rows where the "file url" contained the string "test". From here, the experimenter iteratively filtered the UMAP visualization, unless the participant directed the experimenter to filter the table further. Throughout this process, the experimenter opened files by clicking points in the UMAP visualization, pointing out potential stylistic or structural similarities within clusters in the UMAP, and probing the participant for confirmation or other observations. A cluster was selected and labeled based on agreement between the experimenter and participant. The entire onboarding process took approximately 20 minutes.

After the onboarding phase, the experimenter explained that the participant will now start a search task of their own using Acumen by remote controlling the screen. Participants were informed their task was to add a column to the table of their own design by labeling a region in the UMAP plot. Participants were told the label was an open-ended task and could be anything of their choosing. The experimenter suggested the new column could be within a single project or between projects, capturing high or low level details, or anything else. The label was not required to cover the entire dataset. The only requirement was for the added column to be useful or meaningful to the participant, with a preference for capturing patterns not trivially searchable by filtering the table (e.g., searching for "TensorFlow" in the "Libraries" column and creating a column from those results). This task was chosen to facilitate engagement with all features of the Acumen interface while priming participants to evaluate the relevance of patterns extracted by the UMAP within their own search contexts. The experimenter was available to answer questions, and, when necessary, encouraged the participant to consider information in the Acumen interface to reduce time inspecting the content of source files. Once the participant added a column and was satisfied with the result, the experimenter stopped sharing the laptop screen and directed the participant to complete the study exit survey. Participant searches took approximately 20 minutes. Entire sessions lasted approximately 60 minutes and participants were compensated $20 USD.

Table 2.1: Columns created by participants in our exploratory study. Files without labels automatically labeled as "Unknown" are omitted.

ID	Column Title	Unique Values	Participant-provided Description
P1	`hastransformername`	`LSTM`, `True`, `Transformer_Image`	
P1	`transformerutils`	`True` (Binary)	*"A column that investigates special Transformer models."*
P2	`filetype`	`short tests and configs`, `init`, `inference`	*"determine if a file was doing an inference task"*
P3	`config_file`	`True` (Binary)	*"It captured some config files in NLP models."*
P4	`tutorial`	`True` (Binary)	*"whether a file was used in a tutorial along with being a notebook or notebook like"*
P5	`file_content`	`model_def` (Binary)	*"captured files with transformer model definitions"*

2.7 Results

Results from our exploratory study show how Acumen effectively supported searches and sensemaking in the space of ML projects. We also describe workflows used by participants to discover relationships between projects and source files.

2.7.1 Semi-structured Interviews Reaffirm Existing ML Project Search Challenges

In semi-structured interviews, 4 of 5 participants mentioned using GitHub to search for or implement a recent ML project, and 2 of 5 were given a fixed starting point through their work. One critical result we found in interviews was that, in the past, 3 of 5 participants mentioned finding an initial software repository to use as a template for new work, but then later discovered this initial candidate did not meet their needs. Those 3 participants reported needing to start new searches from scratch with more informed preferences in the course of their own work.

2.7.2 Acumen Helped in Search, Promoted Learning, and Revealed Project Structures

All participants successfully completed the task of adding a column to the ACUMEN table capturing information they discovered in the interface (detailed results in Table 2.1). All 5 of our participants unanimously reported that "Acumen would be helpful when I want to search for ML projects or code". On 5-point Likert scale questions (1 = strongly disagree; 5 = strongly agree), participants also reported ACUMEN was easy to use ($\mu = 4.6$, $\sigma = 0.55$), and that they learned something new about ML projects or code while using ACUMEN ($\mu = 4.4$, $\sigma = 0.55$).

Participants' columns captured semantic indicators

Most of the columns participants created labeled *semantic* (rather than *syntactic* or *stylistic*) similarities between files and on the organization of files among projects. 4 of 5 participants' columns labeled organizational structures of ML projects, capturing config files (P3), model definitions (P2, P5), and training infrastructure code (P1). Participants labeled these organizational attributes within specific domains (e.g., NLP - P3), model architectures (e.g., P5 and P1, who filtered for Transformers), or for narrow sets of projects (i.e., P2, who compared the structures of 3 facial recognition projects). 4 of 5 participants created binary columns to indicate the presence of particular attributes. 2 of 5 participants created columns with multiple values (including a participant who made a multi-value column as an intermediate step towards a final binary column). A summary of the columns and signals that participants identified is shown in Table 2.1.

Acumen helped participants learn about ML projects

In open-ended responses, participants shared what they learned while using ACUMEN, which aligned with their search objectives: *"i learned a lot more about common scaffolding (testing and pipeline and templates)"* (P1); *"tutorial content sometimes comes in raw .py file content as opposed to being in a notebook"* (P4); *"There are many ways to solve the same task using python + any given ML framework."* (P2). This reflects ACUMEN's ability to help users with ML project search as a sensemaking task [135], such that users can iteratively develop stronger mental models of the search space and more refined preferences for their objective throughout the exploratory search process.

Participants used clustering to organize files

When asked what they liked most about ACUMEN, 3 of 5 open-ended responses mentioned the UMAP's ability to help organize files through clustering: *"The clustering of files is very interesting. I feel like near the end of my session I started to understand why files would be clustered together"* (P2); *"really liked the visualization for grouping files"* (P5), *"Cool to link*

back to the GitHub code, and being able to identify clusters of similar file structures [...] I was able to find notebook and notebook-like files very easily from the projection." (P4).

UI and Performance Improvements

When asked what obstacles were encountered while using Acumen, 3 of 5 participants mentioned UI and performance issues that impacted usability: *"a more obvious sign to indicate whether the query is in process or it is finished."* (P3); *"Tooltip occasionally getting in the way"* (P4); *"Speed of the interface was slow"* (P2).

In addition, P2 wished to see a feature that saved a snapshot of particular UMAP views, rather than recalculating UMAP every time a filter is removed (*"Being able to save filters would have made me a lot more comfortable in exploring alternative searches."*). In the current implementation of Acumen, only indices are preserved at every filtering stage, and the UMAP is recalculated when a step is undone. While the recalculated UMAP may be similar, it may be visually different (e.g., rotated or warped). Caching the UMAP calculation for each filtering step will be considered for future work. P3 felt they could explore the search space more deeply with a 3D UMAP option, which could add more separation to clusters (*"files from different semantic areas get grouped together because the plot is limited to 2D"*).

2.7.3 Combining Metadata in Table with UMAP was Important for Effective Searches

All participants in our study used both the table and UMAP as part of their search process. Most participants used the table to initially narrow the UMAP to a set of files broadly within their search scope (e.g., eliminating extraneous ML tasks or only considering projects with a particular dataset). From there, participants progressively filtered the UMAP to discover the higher-level structure of the search space, or to identify relationships between apparent clusters of files.

Acumen table and UMAP both used in search processes

4 of 5 participants mentioned both the table and UMAP when describing the most important part of their overall search process was, e.g.: *"performing an initial filtering to narrow the area of ML I was interested in (attention) followed by finding a relevant grouping of files on the visualiztion"* (P5); *"Trying to understand what each cluster represented, and honing my filtering down to the parts that I wanted (looking for inference or main method code)"* (P2). In particular, P4 described how using Acumen revealed additional dimensions in their search for tutorial notebooks: *"Filtering by extension gets some of the way there but the results would ultimately not be the same as what was given from the system, which captured a broader set of useful examples"* (P4).

Many participants relied on tooltip data which annotated points in the UMAP when hovered over. Tooltips displayed table data (e.g., file name and path, framework, domain)

in context, which provided lightweight indicators and "scents" that signaled the presence of commonalities between files. This was a critical interaction that promoted sensemaking, discussed further in subsection 2.8.2.

Acumen expanded participants' search capabilities

We asked participants if they could write a parser that could have created the column they added to the table. Only 2 of 5 participants said they could, one estimating it would take a few hours, and the other estimating about a day, for what they were able to label in approximately 20 minutes. However, both participants remarked that their parsers may not be perfect, e.g., *"It would be a mixture of regular expressions looking for main methods that perform common ML framework inference methods. This would miss any files that don't use the common API methods "* (P2). Altogether, these results show how having both search (through filtering high-level attributes in the table) and navigation (through exploring relationships in the UMAP plot) was key to effective searches.

2.7.4 Two Key Workflows Emerged

Two key workflows emerged from our participants. 3 of 5 participants annotated structural information in a small area, and zoomed out, exploring related nearby points in the UMAP. The second workflow, used by one participant, relied on combining multi-value columns into a highly specific binary column. These workflows are described in the following sections.

Filtering Down and Propagating Up

A common technique among our participants helped facilitate exploration and discovery by showing how patterns identified in a small set of projects manifested in a broader set of projects. In this workflow, participants first used the table and/or the UMAP to refine the selection to a small set of projects or points in the dataset. It was important for this filtered set to have a relatively small size to accentuate the similarities between points, and thus make them easier to label in a new column. Once a collection of interest was labeled, the participant progressively had filters undone, and set the color coding of the UMAP plot to the newly added column. This incrementally broadened the scope of the search at each step, yet bubbled up indicators identified by the added label.

At each step when filters were undone, the key interaction employed by participants was to explore surrounding points by controlling the `n_neighbors` parameter of the UMAP. This parameter "effectively controls how UMAP balances local versus global structure" [26], i.e., low values will accentuate finer, local differences between files (e.g., syntax and style), and higher values will show higher-level differences (e.g., problem domains and libraries). Participants were able to identify how patterns in small sets generalized to the broader picture in the dataset by controlling the granularity of similarity in the UMAP visualization. For example, P1 tagged files in a set of 3 specific projects, and used this workflow to identify

other projects that were organized in similar ways (using higher values of `n_neighbors`; and files from other projects that performed similar tasks (using smaller values).

Combining multi-value columns to make compound filters

One participant developed a workflow to help produce highly-specific labels in incremental steps. P1 used a combination of the UMAP and table filters to find Transformer models used in computer vision tasks, and then identified a region of infrastructure files for Vision Transformers. In this first step, P1 created a multi-valued column (with multiple labels) to help color-code the UMAP visualization. P1 labeled both positive and negative examples with this column (labeling LSTMs for exclusion, and transformers for inclusion). From here, they zoomed out to the initial view and started a new search for configuration and infrastructure files. Once a cluster was identified, it was zoomed into, and the color coding was switched to the initial column. Within this set of infrastructure files, the points that were color-coded as Vision Transformers were selected and labeled as "Transformer Utils".

While this technique was specific to a single participant, it illustrates how Acumen could be used to generate hierarchical labels. In this workflow, P1 used a temporary column to preserve results from an initial search to be used as a filter in following searches. This was in part to overcome filtering limitations described in section 2.8, but also suggests the value of using Acumen to uncover latent patterns in datasets that may be useful for future searches.

2.8 Discussion and Future Work

2.8.1 Interpretation of UMAP

Acumen uses CodeT5, a large language model, to produce embeddings from software source. Neural networks in general are known to be difficult to interpret [98], and emergent research shows how interpreting large language models can be especially complex [13]. The key interaction of Acumen is to explore an unstructured space of embeddings produced by such a model, and the study task of adding a column to the table requires ascribing a specific meaning to a subarea of the UMAP visualization. Although this task is not an "interpretability task" in the strictest sense (i.e., to identify the *mechanisms* or learned data in a model that contributed to a prediction), it is so in the sense that participants must associate a meaning with unstructured representations of code source files in a visualization. All participants in our study were able to do this, successfully identifying specific areas of the UMAP that were useful in their search task. We believe a key contributor to this result is that labels were not required to generalize to the entire dataset, and could be restricted in scope with table filters. This means labels only needed to capture similarities within local neighborhoods of points, which can be easier to interpret [130, 146]. These local labels were enough to provide useful and meaningful indicators that helped with participants searches. Acumen may be a useful platform to investigate how different practitioners, particularly non-experts, ascribe meaning to learned representations of data in other tasks, such as

language, in future work. One promising direction would be to investigate whether the workflows from our study results generalize and what other workflows appear for other domains, particularly when Acumen is used by domain experts.

2.8.2 Project Search as a Sensemaking Task

Results from our study echo early research results in search user interfaces, where a user study of the Scatter/Gather hierarchical clustering interface showed that the *combination* of navigation and search was a critical factor; omitting either interaction resulted in poorer search performance [61, 127]. At early stages in search, our participants heavily used tooltip data in the visualization (which shows data from a point's corresponding table row) as heuristic signals for interpreting the UMAP plot (as opposed to in-depth examination of source file contents). Many participants skimmed the tooltips to recognize how individual projects were organized using the relative file path field, or identified higher-level similarities between projects using file names. These high-level categorical signals provided important cues and "scents" to indicate whether deeper dives were warranted. There are many opportunities for future work to produce short, meaningful summaries of *entire* ML projects. Recent works have made large strides in summarizing the expected behaviors and known biases of ML models [109] and datasets [41]. Designing effective "tooltips", cards, or just-in-time summaries for code, and unifying those descriptions, would make great strides in aiding in ML project search.

The "Filtering Down and Propagating Up" workflow also complements findings from our semi-structured interviews, where many participants shared a process of searching for GitHub projects to start implementation, but described needing to restart an updated search once they realized the initial project did not fully meet their requirements. Being able to single out a small set of projects, and then find similarly-organized projects or similarly-implemented files may help ML application developers consider and compare alternatives beyond the scope of their initial searches in the same search session. We see these results as a reflection of how both the unstructured and structured elements of search are important for comparing and evaluating ML projects, and how facilitating exploration can assist with this sensemaking task.

2.8.3 Usability and Design Improvements

In open-ended feedback in our exploratory evaluation, participants shared feature requests and areas for UI/UX improvements that could be made in future work. These are described in section 2.7.2. Other additions to future versions of Acumen could including tracking files that were already visited [61] and multiple types of point highlighting (e.g., color plus opacity). Supporting exclusion filters in the table (i.e., "does not contain" filters) could also help facilitate deeper exploration, but is not currently supported by the Plotly DataTable API[9].

[9] https://dash.plotly.com/datatable/filtering

Future works should also investigate alternative visualizations and means for summarizing code embeddings. Using clustering with embeddings to compare image models, a technique described in chapter 4, could potentially be applied to comparing ML projects.

2.9 Conclusion

In this work, we described the challenges of searching for ML projects and presented Acumen, a data extraction pipeline and interactive web application that supports users in searching for ML projects. Acumen supports inspection of *structured* attributes of ML projects, such as datasets and frameworks used, while enabling exploration and comparison of their *unstructured* attributes by browsing source file embeddings in a UMAP visualization. When interacting with Acumen, users iteratively narrow down a large dataset of open source ML projects by refining queries for structured attributes while filtering into meaningful regions and correspondences between files in the UMAP visualization. In an exploratory evaluation, we showed how Acumen can help users who are proficient in software engineering and familiar with ML concepts conduct searches for ML projects. Acumen also helped our participants learn new things about ML projects while expanding their search capabilities. We additionally discuss two key workflows which emerged in our study that enabled participants to effectively browse project structures among the Acumen dataset.

Chapter 3

UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior

3.1 Introduction

Deep Neural Network (DNN) models have been an enabling factor of many powerful ML applications, which can extract and discriminate features from raw data by using massive amounts of learned parameters [18]. These "Deep Learning" (DL) approaches are incredibly powerful, even surpassing human-level accuracy on some tasks [55]. DNNs also enable many new interactions over "Classical ML", such as generating high-dimensional data [50], and supporting *transfer learning*, where selected parameters from a DNN may be retrained to generalize to new applications, creating high performing models without needing millions of data points or massive computational resources.

Non-expert ML programmers, such as software engineers, domain experts, and artists, can use transfer learning to create their own models by using recent frameworks which make this task more approachable with high-level APIs [24, 76]. However, when bugs are introduced, the default failure mode of DL programs is to produce unexpected output without explicit errors [156]. ML novices often expect models to behave as APIs, and have limited mental models to facilitate debugging, sometimes even abandoning ML approaches altogether when they fail [21, 64]. Further compounding the issue, DNNs are considered "black box" models, and cannot be debugged with traditional means such as breakpoints. Experts rely on their experience and tools such as Tensorboard [1] and tfdbg [22] to begin inspecting model behavior, but often fall back on trial-and-error approaches guided by intuition [21]. Adding structure to the DL development process through explanations and guidance could help users close this debugging loop and bridge theory with practice [6, 123].

We introduce Umlaut, the Usable Machine LeArning UTility, a system which uses a multifactor approach to assist non-experts in identifying, understanding, and fixing bugs in their

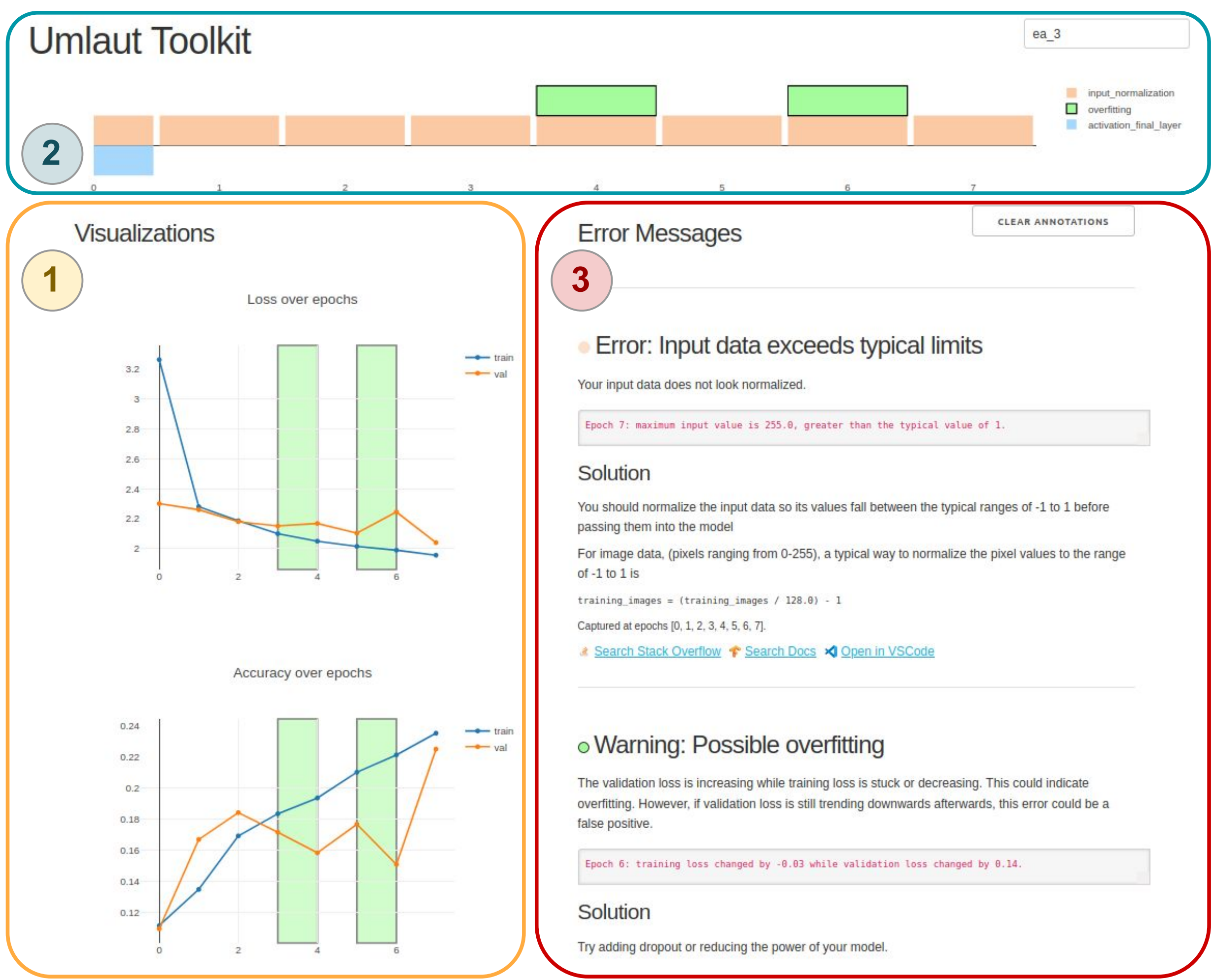

Figure 3.1: The UMLAUT web interface combines visualizations of model metrics (1); a timeline showing errors over epochs (2); and explanations of underlying error conditions with the program context and suggestions for best practices with code examples (3). Plots and the timeline are automatically annotated with with relevant data when errors are clicked.

DL programs[1] (Figure 3.1). UMLAUT draws inspiration from tools and metaphors in software engineering which inspect code to provide warning messages and suggestions to developers. This includes linting [82], unit testing, dynamic analysis [113], and explanation-based debugging [93]. UMLAUT attaches to the DL program runtime, running heuristic checks of model structure and behavior that encode the tacit knowledge of experts. UMLAUT then displays results of checks as error messages that integrate program context, explain best practices, and suggest code recipes to address the root cause(s). Our aim is not to define new heuristics or outperform experts, but to show how existing heuristics used by experts can be automatically checked and made accessible to a broader set of users.

A key objective of UMLAUT is to support users in overcoming three critical "gulfs" of the DL debugging process: mapping from symptoms to their root cause(s), choosing a strategy to address the underlying problem, and mapping from strategy to concrete code implementation. UMLAUT uses an automated checking infrastructure to detect errant model behavior and raise error messages reflecting the surrounding context. Error messages are presented in a web interface that tightly couples errors with visualizations of model output, linking root causes to their symptoms. Error messages include descriptions of their underlying theoretical concepts, and suggest potential debugging strategies to bridge theoretical and practical knowledge gaps. To translate these strategies into actionable changes, UMLAUT errors include code recipes which implement suggested fixes, outbound links to curated Stack overflow and documentation searches, and links to the suspect lines of code in source.

Our work makes the following contributions:

- A discussion of opportunities for supporting the DL debugging process, in contrast to Classical ML, through novel user interfaces
- A novel approach of encoding expert heuristics into computational checks of DL program structure and DL model behavior
- The UMLAUT system, a tool which implements several automatic checks to assist in finding, understanding, and fixing bugs in Keras programs
- An evaluation which shows UMLAUT helps non-expert ML users find and fix significantly more bugs in DL applications.

3.2 Background: Challenges in Deep Learning (DL) Development

The recent success of deep learning in a variety of domains has led to an increase in users of DL, and a corresponding growth of tools that have emerged to help developers with DL workflows. This section summarizes background information and design considerations for tools that aim to aid the DL development process.

[1] Source code for our system is available at https://github.com/BerkeleyHCI/umlaut

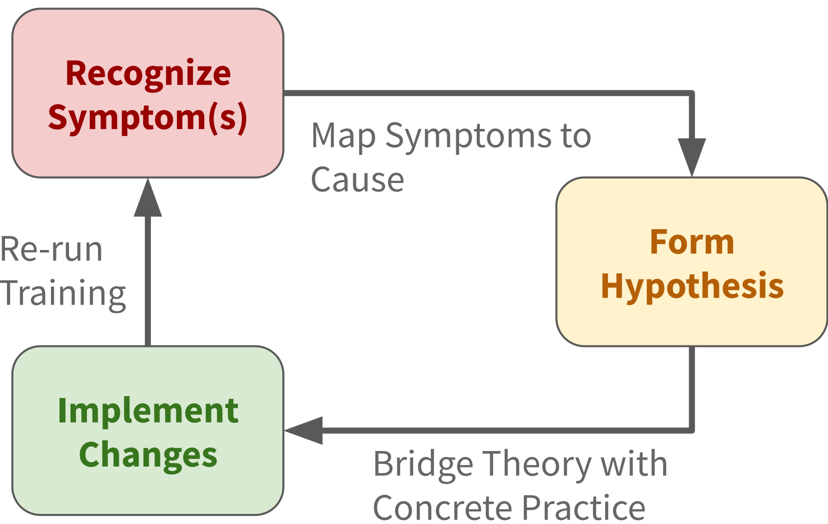

Figure 3.2: To debug DL programs, users first recognize symptoms from errant model behavior or code structure. Experts use mental models built from experience to translate from these symptoms to hypotheses of underlying root causes. Finally, code changes are implemented to test the underlying hypotheses, and training is rerun to check them.

3.2.1 Key Differences of Designing for DL over Classical ML

Both "classical" and "deep" ML development processes are often exploratory [123], where the data, model, and scaffold code are iteratively refined to reach target benchmarks [64]. However, there are critical differences between the implementation of classical ML and DL approaches which significantly alter the developer experience. While classical ML can be effectively applied to many problems, DL can handle high-dimensional, unstructured input and output spaces, such as object detection and audio-cue detection. We characterize the fundamental differences between classical ML and DL in this section and introduce a unique set of challenges that DL support tools should address.

Data Requirements: Both DL and classical ML models require ground truth labeled data to train. However, DNNs often require significantly more data: a rule of thumb suggests a minimum of 5,000 samples per class [49], while classical algorithms such as SVM or Random Forests require far fewer data points. Handling large-scale datasets drives up costs for data collection and processing, particularly in domains with noisy or incomplete data [106].

Featurization: Classical ML algorithms require hand-engineered features to maximize signal from input data. In contrast, DNNs learn features from patterns in the data directly, eliminating the developer-driven feature engineering step [6, 49]. While this provides DNNs

tremendous flexibility in handling unstructured input data, this offers less control and means developers cannot verify whether the model has received "features" from extraneous patterns in the data that confound the effectiveness of the model.

Interpretability: A key feature of classical ML algorithms are that they are often more interpretable than DNNs. While there are many meanings and subsets of model interpretability, it is widely accepted that we do not yet fully understand the exact rules and features that DNNs rely on to produce specific outputs, and how well DNNs generalize to new problems [98]. This makes pinpointing the exact source of numerical errors in DNNs very difficult and gives rise to "silent errors" in the model that can only be spotted by experts with experience pattern-matching code smells to possible errors [178]. Visualization has been a critical tool in interpreting DNN behavior, but this still remains an open research question [69]. In contrast, some classical approaches intrinsically attribute the hand-engineered features most relevant to any prediction.

Training: Unlike classical ML algorithms, DNNs require nonconvex optimization of a large number of parameters. This requires proper initialization of neural network weights [55] and an involved search process for network hyperparameters [12]. DNN training time can take days or weeks, often even requiring online tuning in order to converge [150], lengthening the feedback loop. Experts rely on experience to determine an 'educated guess' of the typical range of hyperparameters which can drastically decrease the search space. Novices encounter difficulty in this process, especially when it generates unknown or ambiguous symptoms.

Transfer Learning: DNNs allow developers to reuse the "feature-picking" parts of the NN, and "fine-tune" the bottom layers to use those feature for new domains and applications. A common interaction is fine-tuning a model trained on many images to a new, smaller, dataset.

3.2.2 Detecting Errors during DL Training and Evaluation

To show how Umlaut fits in the DL development process, we identify four high-level stages of DL development from prior work [6, 122]: (1) Data Processing, (2) Training and Tuning, (3) Evaluation; and (4) Deployment. We focus on the challenges that DL developers face in Phases (2) and (3).

Typical DL workflows require developers to iteratively train and evaluate their models to identify bugs and modeling issues [6, 156]. We characterize this debugging process using the *DL debugging cycle* shown in Figure Figure 3.2. During this cycle, developers repeatedly train models with a specific experimental setup of network architectures, loss functions, and hyperparameters. The model performance is then evaluated by qualitatively inspecting the classification results of various data examples, and quantitatively by calculating accuracy on a validation set. Using the results generated by the training run, developers recognize symptoms, form hypotheses to the root causes of problems, and make decisions to modify the experimental setup using their theoretical understanding of the models. They will then rerun the experiment and this cycle continues until developers obtain a model with satisfactory performance.

Debugging DL models is challenging because even though errors occur in both the training and evaluation phase, the symptoms often only materialize in the evaluation phase in the form of poor model performance [6]. While experts often rely on a continuously refined set of best practices that pattern-match model outputs to effective modifications, novices often think of DL models as black boxes and can have difficulty in recognizing and understanding symptoms [6, 21, 64].

3.2.3 Mapping Symptoms to Root Causes

One critical step in the DL debugging cycle (Figure Figure 3.2) is to map modeling issues from symptoms to their root causes. This step requires developers to analyze model outputs and training curves, classify specific issues from these statistics, and convert them into actionable items. Current error mitigation practices are often ad hoc, such that developers usually only have tools that document performance metrics and general theory resources, but are required to manually draw connections between them. For example, a developer might need to consult best practices collected from literature, expert blogs, and academic lectures [14, 87, 89, 144] to derive a set of actionable items that resolve their issues. Developing this skill requires extensive time and exposure to errors at different stages and levels of abstraction: the program, theory, data, etc. These skills are essential for successfully training a model with high performance, yet helping novices gain the tacit knowledge needed to successfully diagnose and debug model issues remains an open challenge [7].

3.3 Related Work

We map prior work in three axes that correspond to Section 3.2 based on their contributions, and discover design opportunities for Umlaut in complementary areas.

3.3.1 Interfaces for Supporting Classical Machine Learning Workflows

HCI research has produced novel interfaces which allow users to interactively train and tune ML models as early as 2003 [33, 37, 90, 101]. Gestalt is a toolkit which adds structure to the ML development process with an IDE [122]. Makers can use ESP to interactively train and deploy gesture recognition models on hardware [103]. Other works help compare DL model performance, but only once the models are trained [112]. While these tools support the feature engineering workflow required for classical ML, Umlaut focuses on training and tuning DNNs. DNNs instead learn features from input data and enable powerful new applications.

3.3.2 Tools for comparing and improving DL Model Performance

Research and engineering teams have produced novel interfaces to compare model performance [112, 159, 172] and subsequently debug modeling issues. Because of the intrinsic relationship between training data and a model, these tools can highlight relevant training data contributing to outliers [67, 120, 165] and refine the model itself [4]. Taking steps towards debugging these issues, TensorFuzz adapts coverage-based fuzzing to identify model inputs which generate numerical errors [117].

In addition, UMLAUT is inspired by a field of academic research in Explainable Artificial Intelligence (XAI) which help practitioners *interpret* the output and behavior of their ML models. DNNs often have too many parameters to easily understand, and explaining their output is an active area of research [44]. Methods like Saliency Maps can highlight the specific parts of an input image used to make a prediction [86, 118], while Concept Activation Vectors (CAV) can explain the higher-level concepts used [91].

Evaluating the performance of ML models is a critical step, but all of the aforementioned prior work depends on having an already-trained model. UMLAUT assists users in the training step required *before* evaluation. We believe UMLAUT is an early step in both debugging and providing explanations of neural network output during the training process.

3.3.3 Prescribing Best Practices and Code Changes in Context

As mentioned in Section 3.2, current tools mostly help inform code changes in DL development workflows by tracking and instrumenting experiments for large-scale deployments [40, 77, 97]. While these tools are critical for developers to track the progress of their experiments, they typically do not directly report any potential errors. ML practitioners can also add instrumentation and visualizations to their DL models using toolkits such as TensorWatch [142] and Lucid [118], but the choice of visualization and its interpretation requires expertise.

Several studies conduct empirical analyses of bugs found in ML programs using data from Stack Overflow and GitHub [73, 79, 176, 178]. These works create a high-level classification of common bugs, but don't link between symptoms, root causes, and actionable items in context. On the other hand, some tools in research [14, 139] and deployment (such as EarlyStoppingHooks [24]) use algorithmic checks for training. However, while these actions are taken in context during training, they do not produce error messages, link to root causes, or tie back to other information (e.g. learning curves).

Inspired by work in supporting traditional software development [16, 39, 48, 53], UMLAUT also suggests code examples from official documentation and best practices pulled from StackOverflow, which helps users to directly address errors and dive deeper into the code. UMLAUT builds upon established paradigms in software engineering such as linting [82], unit testing, dynamic analysis [113], and explanation-based debugging [60, 93]. UMLAUT works *in context* to help users interpret the behavior and inspect the points of failure of their ML applications [58], as similar paradigms have not been extensively explored for DL develop-

ment. We draw additional inspiration from software visualization [149] and tutorial systems for complex user interfaces [51]. UMLAUT also adapts an automated-checking infrastructure that enables running tests over model runtime behavior to flag problems for non-expert users. This approach has been used in other HCI research to assist debugging electrical circuits and embedded systems [32, 102].

3.4 Debugging ML Programs with Umlaut

To use UMLAUT, users attach a UMLAUT client to their program, which injects static and dynamic heuristic checks on the program, parameters, model structure, and model behavior. Violated heuristics raise error flags which are propagated to a web-based interface that uses interlinked visualizations, tutorial explanations, and code snippets to help users find and fix errors in their model. UMLAUT also emits flagged error messages to the command line, inline with Keras training output, to reduce context switching. Heuristics, errors, and their implementation are described further in Section 3.6.

To illustrate how UMLAUT works in practice, consider Jordan, a park ranger who wants to receive a notification when rare birds appear in a bird feeder camera. Jordan has domain expertise in ornithology and birding, and has taken an online data science course, but they are not an ML expert. Jordan was able to prepare a labeled dataset of birds at the feeder using previous recordings, and they found a template project from the data science course to use a pretrained Resnet image classification model [54] for transfer learning. Jordan's next step is to fine-tune the pretrained model on their new dataset. After fixing the input image shapes from a bug produced by Keras, Jordan is able to get the training loop to run. The loss is now decreasing, and accuracy rising, but only to 60%–not enough for their application. Jordan manages to contact their former data science instructor, who volunteers a quick look at the program, but can't seem to find anything wrong. Jordan hears that UMLAUT can help detect and fix bugs in DL programs, and gives it a try.

3.4.1 Importing Umlaut and Creating a Session

To use UMLAUT, *Jordan adds three simple lines of code to import and attach it to their program: they import the* UMLAUT *package, pass the model and other inputs to the* UMLAUT *object, and add the* UMLAUT *callback to the training loop.*

A key design principle of UMLAUT is to ensure it integrates smoothly into existing DL frameworks and development tools. We choose to integrate UMLAUT into the Keras API of Tensorflow 2, because of its high-level API and its broad community support. At runtime, the UMLAUT client adds a callback and injects shims into the Keras training routine. While the model is training, the client runs several heuristic checks, sending metrics and raised errors to a UMLAUT server through a named session. Colliding session names are appended with an auto-incremented integer.

3.4.2 User specification of Umlaut checks

Before running UMLAUT *with their training loop, Jordan tells* UMLAUT *that their model expects images as input and a sparse vector output indicating the predicted class by passing the* `inputtype='image'` *and* `outputtype='classification'` *arguments to their* UMLAUT *call. These flags tell* UMLAUT *to run additional checks (e.g., ensuring the input dimensions are consistent with image formats and that a softmax layer is used on the output).*

Users can supply arguments to UMLAUT which specify the expected input and output formats of the model, reflecting the high-level problem statement. UMLAUT supports image or sparse text inputs, and classification or regression outputs. Depending on the user's guidance, UMLAUT selects different checks to run based on the input, and alters the content of output error messages (e.g., ensuring an RGB color image has 3 dimensions and is normalized, or that a classification loss function such as cross entropy is not used for a regression output). This specification is optional, but leads to more detail in error messages and a wider selection of checks. This is a novel interaction for DL debugging, and can be used to ensure the model architecture and data preparation match with the intended problem type.

3.4.3 Actionable Error messages

When Jordan runs UMLAUT *with a training session, they see some errors appear in the web interface. They first turn to an error marked as Critical ("Missing Activation Functions").*

A significant, novel component of the UMLAUT system is that it generates error messages to *explain* silent error conditions. UMLAUT lists suspected DL program issues, highlights their root cause(s) with integrated program context, offers potential solutions from collected best practices, and directly links error messages to visualizations of model output. Error messages produced by UMLAUT contain the following elements:

Title and Severity Qualifier

Error messages produced by UMLAUT have titles which reflect their respective root causes. Titles are given severity qualifiers (*Warning*, *Error*, and *Critical*) depending on the expected impact on model performance. Warnings have minimal impact on accuracy, but may lead to issues in the future (e.g., an issue with validation data). Critical errors can prevent the model from learning from data at all (e.g., a hyperparameter causing loss to reach NaN). Severity qualifiers are added manually to error message titles, but future iterations of UMLAUT could automatically assign them based on predicted impact.

Instructional Description with Program Context

Studies of the experiences of non-expert ML developers show that building an understanding of ML theory and bridging that theory with practice are significant hurdles [21, 64]. In UMLAUT error messages, descriptions explain the surrounding ML theory, describe the

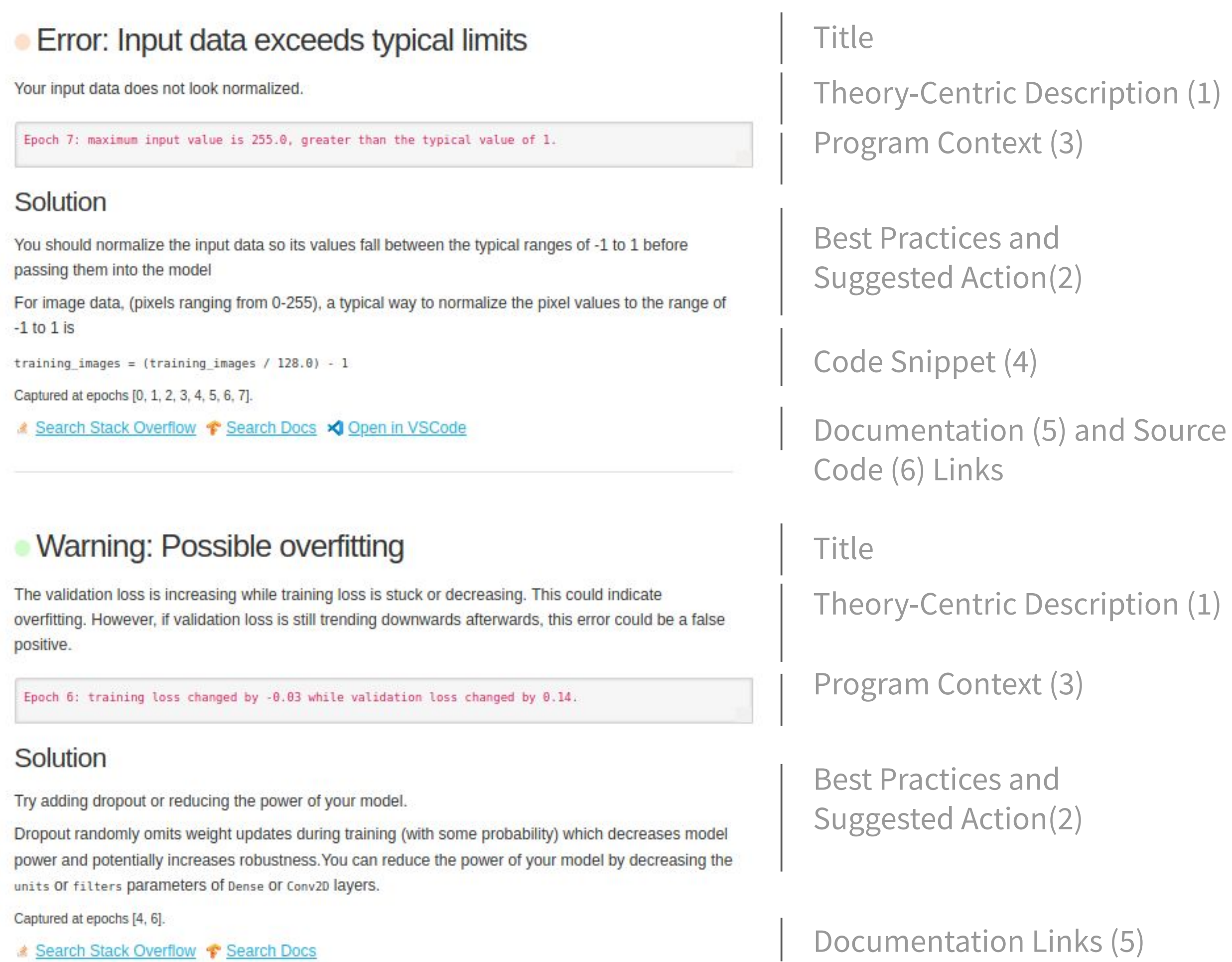

Figure 3.3: UMLAUT errors include several elements to help developers close the DL debugging loop. Errors include short and long descriptions (1) with suggested solutions (2), often incorporating program context (3). Solutions can include code snippets or hints (4), and outbound documentation and Stack Overflow links (5). To help users pinpoint the root cause(s) in code, some errors include links to open the source file in VSCode at the specific location of the suspected root cause (6).

heuristic check used to raise the error, and suggest actionable bug resolution steps in order to bridge knowledge gaps for non-expert users. Error messages can also include *program context* to shed light on the particular conditions which raised an error during program execution. The context is dependent on the particular error, and includes runtime data, such as values of variables which exceeded the limits of a heuristic, prototypes of API calls with invalid arguments, or names of model layers with invalid hyperparameters assigned.

Jordan remembers learning about different activation functions for DNNs in their class, helped by the quick refresher from UMLAUT*'s error description. Jordan looks at the program context in the error and sees that* UMLAUT *printed the names of layers in the model with linear activation functions—the bottom layers which were swapped in for transfer learning on the bird dataset. It was a simple mistake: Jordan simply forgot to add an activation argument, and Keras assigns linear activations when the argument is omitted.*

Bridging to Best Practices with Code Examples

While theory is critical for building mental models to aid in DL debugging, theory alone is not enough to guide users in decision making when debugging. Furthermore, understanding the proper API usage of DL frameworks themselves can remain challenging to novices [64]. UMLAUT makes error messages actionable by including descriptions of potential solutions based on best practices and by instantiating them with concrete code examples.

Beyond code snippets, error messages in UMLAUT can provide outbound links to curated Stack Overflow searches (e.g., `[keras] is:closed from_logits` to search for closed issues with a "Keras" tag for a search query) and links to Tensorflow documentation for relevant APIs. Altogether, program context, grounded in explanations of *why* it has raised errors, helps develop user mental models of DL debugging; while code snippets embodying best practices help users close the debugging loop by making the appropriate fixes to their application.

Jordan remembers learning about many kinds of activations in their class—sigmoid, tanh, relu, . . . —but can't remember when to use which one. Reading further in the UMLAUT *error message, a code hint suggests adding activation='relu' when working with image data. Jordan copies this hint to paste into their program.*

Referencing the Suspected Root Cause in Code

To further assist users in closing the debugging loop in larger models or more complex programs, the UMLAUT client ingests the source of the program being debugged and inspects stack frames in the Python runtime to guess the closest line of code to the source of a given error. The UMLAUT web interface renders these links as URLs which open the Visual Studio Code editor to the specified line and character in the file where the bug occurred.

Jordan notices the error message has an "Open in VSCode link". They click the button and are taken directly to first layer missing an activation function. They paste the code hint

from UMLAUT *there and into the other layers missing nonlinear activations. Relieved it wasn't something more serious, Jordan restarts the training process.*

3.4.4 Bidirectional Link Between Errors and Interactive Visualizations

Inspired by development tools such as Tensorboard [1] and Weights and Biases[2], UMLAUT uses simple visualizations to show how model training progresses over time. Line plots show loss and validation accuracy values at the end of every training epoch (a complete iteration over the training dataset), with multiple traces for training and validation. As a rule of thumb, decreasing loss and increasing accuracy plots that slow over time are a positive indicator. When errors are present in a DL program, anomalies may appear in these plots, which are often subtle and require expertise to decipher [89].

Jordan keeps an eye on the UMLAUT *plots as new training metrics stream in. They notice the validation loss plot starts decreasing, then plateaus and starts increasing. A new warning message pops up: "Possible Overfitting". Clicking the error highlights the epochs in the loss plot where the validation curve started increasing while the training curve decreased, confirming Jordan's suspicion that this was an undesired result. Following the recommendations of the overfitting warning, Jordan adds Dropout to the model and reruns training.*

Error Timeline

UMLAUT also displays a *timeline* visualization, which encodes the type and frequency of errors encountered in the DL program over time. For every training epoch, unique errors are stacked on a vertical axis, distinguished by a 4-element categorical color scale. This visualization allows the user to inspect the behavior of the model and training process over time, e.g., spotting errors flagged before the beginning of the training process (plotted below the horizontal axis) or errors which only appear later during training (such as overfitting or spiking loss from an outlier in data). Users can click on the timeline or on error messages to highlight specific regions in the line plots. Plot annotations show which epoch(s) the errors occurred, and where the behavior of the curves caused a heuristic to raise an error. Inspecting the timeline may also help determine when a raised error was a false positive, e.g., when an error appears sporadically, or rarely.

After the last training run, Jordan keeps an eye on the loss and validation plots. They seem to look fine this time, but the Overfitting warning pops up again. They're skeptical, since they just implemented a fix earlier, so they click the error to highlight parts of the error timeline, and the loss and accuracy plots. Jordan sees there were two epochs when the validation loss went up a small amount, but the overall trend looks fine. They make the judgment call that the error was likely a false alarm, and save the model checkpoint, at an accuracy of 84%.

[2] https://www.wandb.com/

With the model trained, Jordan writes a quick program that uses it to classify live images from the camera feed and notify them by email when a rare bird appears. The system not only helps Jordan enjoy the wildlife, but logging the rare birds' feeding activity from the classifier output also helps in their conservation efforts.

3.5 Umlaut Heuristics

In order to codify best practices from experts into UMLAUT's automated checking infrastructure, we identify and implement 10 preliminary heuristics based on commonalities in various sources including lecture notes [89, 115, 144], industry courses and articles [70, 99], textbooks [49], expert practitioner blogs [87], default values in APIs [1, 24], and early-stage research cataloguing tensorflow program tests [14]. We prioritized heuristics which covered bugs and conceptual misunderstandings shown to be common themes in Stack Overflow questions, open source DL projects, and expert interviews from existing literature [73, 176, 178].

Our heuristics map to common issues in data preparation, model architecture, and parameter tuning. We implement a check for each which is *static* (using a snapshot of the program prior to training) or *dynamic* (analyzing the program during model training runtime). Each heuristic check has an associated severity qualifier *(Critical, Error, Warning)* and error message written by the authors. These error messages include context and suggestions summarized from the heuristic's sources. Our list is not exhaustive, and we discuss how UMLAUT may be extended with additional heuristics checks in Section 3.6.

Although the heuristics we select are widely-accepted and often apply to common use cases, they may not always apply to a user's specific context (resulting in a false positive or negative). In particular, some heuristics used in the ML community suggest concrete values, e.g., for learning rate or image dimensions. UMLAUT adopts commonly used values, e.g., input normalization between -1 and 1. There values might change with new developments in underlying algorithms or community conventions. Future versions of UMLAUT could have such values exposed as configuration parameters that users can update over time.

3.5.1 Data Preparation

Input Data Exceeds Typical Limits (dynamic)

Normalizing input data to a common scale can help models converge more quickly, weigh features more evenly, and prevent numerical errors [99, 144]. Normalization is often regarded as an important "default" setting [156]. UMLAUT checks if the input data exceeds the typical normalization interval of $[-1, 1]$.

NaN Encountered in Loss or Input (dynamic)

The loss value of a training batch can overflow and become NaN during training as a result of non-normalized inputs or an unusually high learning rate [89, 99]. Umlaut checks whether NaN values appear in the loss output, and, if so, whether they appear in the input. Umlaut separately checks if the current learning rate is unreasonably high (Section 3.5.3) which could also be causing NaN loss values.

Image input data may have incorrect shape (dynamic)

DL frameworks expect image inputs to convolutional layers to follow a particular format (typically "NHWC" or "NCHW")[3]. If these dimensions are not ordered as expected, the program may still run without an error, but the network will have incorrect calculations of convolutions in those layers (i.e. convolving over the wrong channels). This can reduce accuracy and speed due to a resulting incorrect number of parameters. Umlaut checks the input sizes of these dimensions (assuming input image height matches width, common for many vision tasks) against the configured ordering. Umlaut raises an additional error message if the configured channel ordering is not optimal for the hardware the program is running on (CPU or GPU).

Unexpected Validation Accuracy (dynamic)

When a model's prediction accuracy on a validation set is unusually high or when it exceeds the value of its training set accuracy, this may indicate leakage between the training and validation data splits [99]. Umlaut checks if the validation accuracy exceeds the training accuracy or exceeds 95% after the third epoch (to reduce noise).

3.5.2 Model Architecture

Missing Activation Functions (static)

When multiple linear, or "Dense" layers are stacked together without a non-linear activation in-between, they mathematically collapse into a single linear layer rendering additional parameters useless. Therefore, a nonlinear activation function must be used between them to produce nonlinear decision boundaries [49, 144]. Umlaut inspects the model architecture and raises an error if two linear layers are stacked together without a nonlinear activation in between.

Missing Softmax Layer before Cross Entropy Loss (static)

Some loss functions expect normalized inputs from a softmax layer (i.e., classification outputs that model a probability distribution, such that each class's probability is between 0-1 and

[3] https://www.tensorflow.org/api_docs/python/tf/nn/conv2d

sums to 1) [156]. The Keras defaults for cross-entropy loss expect softmax inputs, so omitting a softmax layer (or omitting the `from_logits=True` argument to the loss function) can result in a model that learns inefficiently due to improper gradients. UMLAUT checks that softmax is being calculated before the cross-entropy loss calculation.

Final Layer has Multiple Activations (static)

A complementary problem to a missing softmax layer prior to the loss calculation is the addition of an extra activation function. UMLAUT checks for stacked activation functions, which is redundant or may even impact model performance negatively.

3.5.3 Parameter Tuning

Learning Rate out of common range (dynamic)

Setting the learning rate too high or too low can cause drastic changes to model behavior and cause several symptoms in output. A learning rate which is too high can cause NaN outputs or a non-decreasing loss value during training, while a low learning rate can cause loss to converge to non-optimal values early [89, 156]. Best practices for initializing learning rates vary: Keras initializes the Adam optimizer with a learning rate of 0.001, while some experts suggest 0.0003 [87]. Because selecting a learning rate is highly problem-specific, UMLAUT checks that the optimizer's learning rate falls between 0.01 and 10^{-7} (near the limit of precision for 32-bit floating point numbers) and raises an error if it falls outside this range.

Possible Overfitting (dynamic)

Overfitting occurs when a model fits training data too closely, reducing its ability to generalize to new data. This is a core challenge to DL development since features created by a DNN may capture subtle elements disproportionately common in training data [98]. To check for overfitting, UMLAUT determines if the generalization error of its model has started to increase while the training error continues to drop, a widely-accepted indicator of overfitting [49, 89, 99, 144]. Our implementation of this check is reproduced in pseudocode below:

```
function DetectOverfitting(epoch, model, logs)
    d_loss = logs.loss - model.history.prev_loss
    d_val_loss = logs.val_loss - model.history.prev_val_loss
    if d_val_loss ¿ 0 and d_loss ¡= 0 then
        raise OverfittingError(epoch, context=(d_loss, d_val_loss))
    end if
end function
```

High Dropout Rate (static)

In order to prevent overfitting and aid in generalization, dropout can be used, which probabilistically prevents a percentage of neurons from receiving gradient updates. UMLAUT checks the model configuration before training and raises a warning if the dropout probability exceeds 50%, which could lead to redundancy in the model and a reduction in accuracy due to lower-than-desired number of parameters. This error is often be caused by the users' confusion between the 'drop' and 'keep' probability, which are opposites. Our implementation of this check is reproduced in pseudocode below:

```
function DetectHighDropout(model)
    flagLayers = list()
    for layer in model do
        if layer is Dropout and layer.dropoutRate ¿= 0.5 then
            flagLayers.append(layer.index, layer.name, layer.dropoutRate)
        end if
    end for
    if flagLayers then raise HighDropoutError(context=flagLayers)
    end if
end function
```

3.6 Implementation

UMLAUT is comprised of 2 major components. The first is a client program which interfaces with a Keras training session, injects checks into the runtime, then uses those checks to raise errors. Metrics and errors are streamed from the UMLAUT client to the second component, the UMLAUT server. The server logs errors and metrics in a database, and renders data and error messages with a web application.

3.6.1 Umlaut Client Shims and Structure

The UMLAUT client is packaged as a Python library which can be imported and configured for use with a Keras program in 3 lines. Users import the library, configure and initialize the imported `UmlautCallback` object which returns a `tf.Keras.callbacks.Callback` instance, and pass that callback instance as an argument to the `model.fit` training function.

In order to access and diagnose a broad range of error symptoms, UMLAUT requires several data sources from the DL program runtime. Because these sources must be transparently instrumented, we refer to the instrumentation as "Model Shims". UMLAUT uses various APIs and shims to ingest the following runtime information (Figure 3.4).

Keras Callbacks Provide Epoch Number and Training `logs`: The Keras framework implements a callback mechanism which provides hooks at various steps during the model training process. The UMLAUT client is primarily implemented as a Keras callback

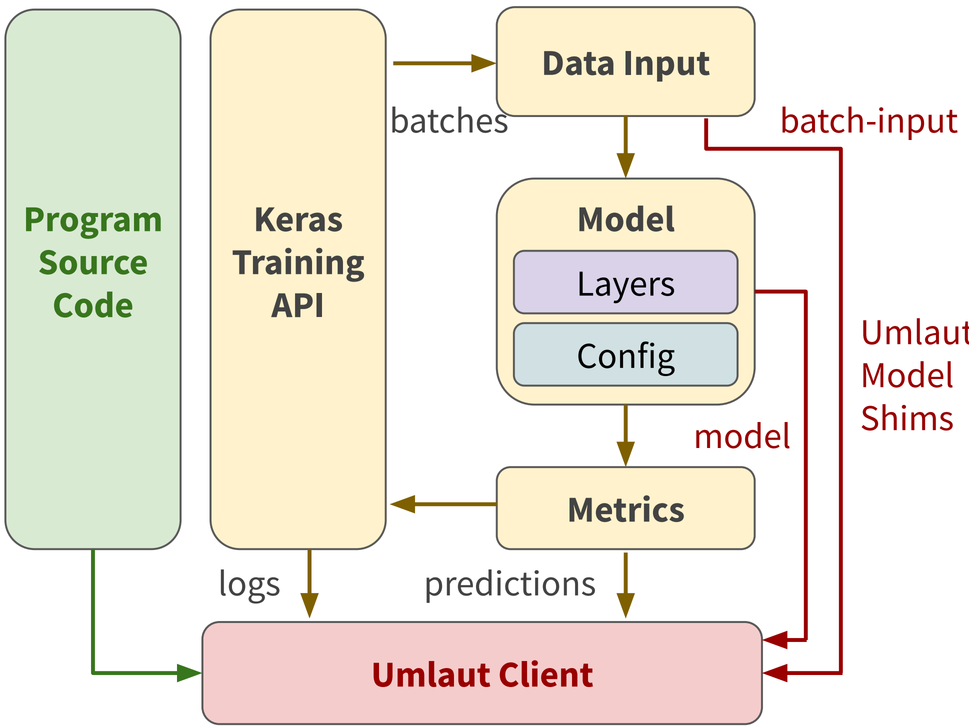

Figure 3.4: UMLAUT uses the Keras callback system to collect metrics about the training process during runtime. UMLAUT also injects variables into the underlying Tensorflow model graph to capture input and output values, and collects a reference to the model object.

which runs static program checks before training starts and dynamic checks after the completion of every training epoch. Pre-training checks are not provided data from Keras, and rely on access to the model object and source module (described below). Callbacks fired during training are passed epoch numbers for indexing, and a `logs` object which contains the loss and accuracy values from the current epoch on the training and validation data.

Users Provide the `tf.keras.Model` Object: When initializing the `UmlautCallback`, Users must pass the model being trained as an argument. The model object exposes many critical elements for diagnosing errors. The `logs` object provided by Keras callbacks only provides a snapshot of the model's loss and accuracy metrics. Having access to the model instance exposes a `tf.Keras.callbacks.History` object which stores loss and accuracy values from every epoch in the current training run. The history object allows UMLAUT to check the *behavior* of the model over time, enabling more complex heuristics (e.g., detecting overfitting). The model object also exposes its underlying structure, e.g., the individual layers and optimizer.

Model `call` Overrided to Access Input and Output: In order to access copies of data passed into and predictions from the model during training, we override its *call*

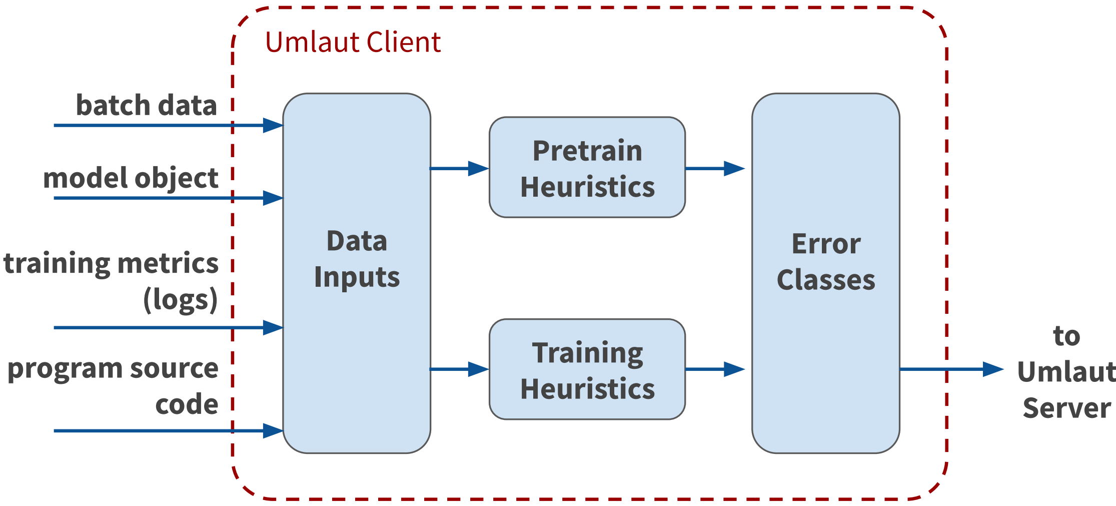

Figure 3.5: The UMLAUT client uses data collected from shims to run static checks of the model before training, and dynamic checks during training. Heuristic checks and errors (reflecting root causes) are distinct concepts in UMLAUT's architecture, allowing similar, yet subtly different symptoms to raise different root causes from within the same check.

function. To do this, we add two `tf.Variable` objects to the model execution graph (before and after). The variables store copies of the model's latest input and output data, and can be evaluated in the Tensorflow session used in the Keras backend.

Module Source Code Captured by Searching Stack Frames: Some error messages rely on the location and contents of the program source code. UMLAUT uses the Python `traceback` library to guess which source module contains the training loop, and then stores the contents of the file for searching.

3.6.2 Umlaut Client Logic: Running Checks and Raising Errors

During the training process, UMLAUT aggregates inputs from model shims and dispatches them to test runners. Test runners run *static* checks before training starts, and *dynamic* checks during program execution, after every training epoch. Checks during either of these stages can raise errors, which include client program context, and are stored on the UMLAUT server. A key design choice in the implementation of UMLAUT was to decouple checks and errors. This allows more flexibility and brevity in cases where one heuristic could detect similar symptoms that map to different errors.

Static checks inspect the structure of the model and its parameters without any context from runtime. Dynamic checks use context from program runtime in concert with model structure and parameters. Dynamic checks can capture *snapshots* of the program execution environment (e.g., to find input data with NaN values), or can track the *behavior* of the

model over time (e.g., capturing overfitting when training loss decreases and validation loss increases). The performance impact of static checks is minimal, and model size impacts performance on the order of ms. For dynamic checks, UMLAUT mostly operates on aggregate metrics already collected by Keras, and the added operations from shims have no noticeable effect on performance. We confirmed this by running UMLAUT on more complex models (see Section 3.7).

When a check raises an error, it initializes that error with the program context necessary to render the error in UMLAUT's web application (e.g., including the names of layers with missing activation functions or the value of a high learning rate). At the end of every epoch, the client sends metrics (loss and accuracy for test and validation sets) and errors to the server. For errors, only a unique error key and the related context is sent to the server, and the server renders the error's static description and contents. Since these requests use aggregate data, they impact performance on the order of tens to hundreds of ms (including network latency) per epoch.

UMLAUT's design allows new checks and errors to be added in a standardized way. To do this, a developer must add a new error message by subclassing a base error template in the UMLAUT server, and a check function that raises the new error to a check runner in the UMLAUT client.

3.6.3 Umlaut Server

The UMLAUT web interface is implemented using Plotly Dash with the Flask web framework, and MongoDB for the database. The web application exposes a REST API to accept updates from the UMLAUT client which stores errors and metrics associated with their session in the database. When a user navigates to the UMLAUT session view, the page polls the database and rerenders the page when new data is present. Interactive graphing features in the web application are implemented using Plotly Dash's Pattern Matching Callbacks feature. This functionality allows click events on an error, the timeline, or a plot to update the other corresponding elements.

3.7 User Evaluation

We evaluate the usability of UMLAUT's interface as well as its ability to help developers find and fix bugs in ML programs in a within-subjects user study with 15 participants. We introduce bugs into two image classification programs, and measure the number of bugs participants find and fix, with and without UMLAUT.

3.7.1 Participants

We recruited 15 participants (12 male, 3 female; ages 18-30, $\mu = 23.8$, $\sigma = 3.1$) from university mailing lists to participate in our study. Through a recruiting survey, we accepted

participants who were at least familiar with ML concepts and development, but who did not identify as an expert or professional (i.e., excluding ML reserachers who primarily develop ML models). Of our participants, 12 had integrated existing machine learning models into projects, and 9 had retrained the last layers of an existing machine learning model to adapt it to a use case. 4 participants had developed new machine learning models, and 2 had contributed to open source machine learning projects. Questions determining expertise were adapted from Cai et al. [21]. 14 participants were graduate students, and 1 undergraduate. 11 had academic backgrounds in computer science, 3 in electrical or computer engineering, and 1 in mechanical engineering. Participants were compensated $20 USD. Evaluations lasted under 60 minutes.

3.7.2 Setup

Due to the COVID-19 pandemic, the study was conducted remotely using Zoom videoconferencing software on the experimenter's laptop, a 2016 MacBook Pro. Participants used the Visual Studio Code IDE with the Pylance Python language server [119] and VS IntelliCode [104], which together provide relevance-ranked autocompletion and syntax checking. Python files for the debugging tasks were loaded and executed by the IDE on a Google Cloud Platform instance with an Nvidia Tesla T4 GPU to reduce model training time. For the CIFAR-10 task, training the provided model for 10 epochs took under 1 minute.

3.7.3 Study Design and Tasks

We modeled the design of our user evaluation after that of Gestalt [122]. Our study was a within-subjects design, comparing UMLAUT to a baseline condition across two debugging tasks. To account for interaction effects from the ordering of these conditions, tasks were counterbalanced by condition (baseline vs UMLAUT) and by order (Program A vs Program B). We measured the number of bugs *found* (i.e., the explicit root cause verbally indicated by the participant) and *fixed* in each task. Bugs which only had a partial fix (e.g., adding missing nonlinearities in convolutional but not linear layers) were not counted as fixed.

For the debugging tasks, we created a simple Keras program which loads the CIFAR-10 dataset [94], constructs a 7-layer convolutional neural network, configures cross entropy loss and Adam optimization [92], trains the model for 10 epochs, and evaluates model accuracy on the CIFAR-10 test set. We designed this program to be as simple as possible—under 35 lines of code (under 40 when adding UMLAUT)—for two key reasons. First, simplicity strengthens the baseline condition by being easier for the participant to fully understand. Second, the model is able to train quickly (under one minute on a GPU) before the test accuracy plateaus around 77%, making more iteration feasible in the study timeframe compared to a larger (but potentially more accurate) model.

We created two modifications of this program, *Program A* and *Program B*, and inoculated both with three unique bugs. These programs both execute without any explicit Python errors or warnings, but the bugs impact the accuracy of the model at different levels of severity:

low (approx. 0-5% reduction in accuracy), *medium* (approx. 6-20% reduction in accuracy), and *high* (accuracy will not increase beyond random chance). The bugs in both programs were also chosen to span common stages of the ML development process: *Model Architecture*, *Parameter Tuning*, and *Data Preparation*. Finally, the bugs may generalize well to different learning tasks, e.g., image classification, sentiment classification, pose estimation,...).

The bugs introduced into Program A were:

- **A1:** No softmax function was added after the final `Dense` layer, causing the optimizer to receive unnormalized logits and not improve loss (High severity, model architecture)
- **A2:** Dropout rate set to 0.8, resulting in only 20% of model capacity being used (Medium severity, parameter tuning)
- **A3:** Input images were not normalized, with values ranging from 0-255 (Low severity, data preparation)

The bugs introduced into Program B were:

- **B1:** Learning rate was set to -1e3 instead of 1e-3, resulting model being unable to learn from data (High severity, parameter tuning)
- **B2:** No ReLU activation functions were added to the model, resulting in stacked convolution or dense layers collapsing into a single layer (Medium severity, model architecture)
- **B3:** Validation data overlapped with the training set, picking the first 100 training images (Low severity, data preparation)

As a test, we connected UMLAUT with VGG16 and ResNet101 from the `Keras.applications` API and ran it in the same scenarios as our user evaluation (adding bugs A1, A3, B1, B3). A2 and B2 were not considered as they would require source code changes to Keras. We verified the same errors as our small test model were raised.

3.7.4 Procedure

After completing an entry survey, participants were shown a minimum-example Keras program which fit a linear model (2 `Dense` layers) on a different, simpler dataset (Fashion-MNIST [168]) in the Visual Studio IDE. Participants were shown the dataset Readme, the structure of the program was explained (imports, data loading, model architecture, training configuration, training, and evaluation), and the program was executed in the editor. For participants starting in the UMLAUT condition, the application had lines of code added for invoking UMLAUT. The UMLAUT web interface was loaded on a web browser on the researcher's laptop, and the example program was run with an UMLAUT session attached. The UMLAUT command line and web interfaces were explained, and participants were told error messages were based on heuristics, so there may be false positives. For participants starting

in the baseline condition, the example program with UMLAUT code added was demonstrated between completing the baseline and UMLAUT tasks.

Before starting the first debugging task, participants were shown the Readme for the dataset used by their debugging task programs, CIFAR-10 [94]. Participants were told they would be shown a program with multiple bugs, and their task would be to find, explain, and fix all the bugs they found in that program. Participants were told they should not need to make major architectural changes to the models (e.g., by adding or removing layers, or changing the sizes of `Conv2D` or `Dense` layers), but were able to if desired. Participants were told they could use any online resources needed, e.g., documentation, Stack Overflow, or web search; and the researcher could troubleshoot the apparatus or explain the UMLAUT interface, but not assist with debugging. Finally, participants were told a bug free version of this program could have a target test accuracy of 77%, but were reminded their goal was not to maximize accuracy, and that a high accuracy does not guarantee a bug-free model. The training period took approximately 15 minutes (plus 5 for UMLAUT).

Participants were then shown program A or B, in the baseline or UMLAUT condition. The only differences between conditions were that UMLAUT code was added to the program and opened in a web browser. Participants were not allowed to use UMLAUT software in the baseline condition. After completing the first task, the other program was shown, in the other condition. Participants were limited to 15 minutes of debugging time per program.

3.8 Results and Discussion

3.8.1 Umlaut Helped Participants Find and Fix Significantly More Bugs

Across both programs, participants using UMLAUT found more bugs ($\mu = 2.8, \sigma = 0.4$) compared to the baseline condition ($\mu = 1.8, \sigma = 1.1$). This difference is statistically significant (Wilcoxon Signed-Rank test, $Z = 2.67, p = 0.004$). Furthermore, participants using UMLAUT were able to implement fixes for more bugs ($\mu = 2.5, \sigma = 0.5$) compared to the baseline condition ($\mu = 1.5, \sigma = 0.9$). Again, this difference is statistically significant (Wilcoxon Signed-Rank test, $Z = 2.65, p = 0.004$).[4]

Furthermore, survey responses collected from participants confirm and strengthen these findings. On 5-point Likert scale questions (1= strongly disagree, 5=strongly agree), participants indicated that UMLAUT helped them find ($\mu = 4.3, \sigma = 0.70$) and fix ($\mu = 4.0, \sigma = 1.1$) bugs they would have not noticed without it. Participants also indicated a high likelihood of integrating UMLAUT as a regular part of [their] ML development processes ($\mu = 4.3, \sigma = 0.60$). The distribution of ratings for these questions is shown in Figure 3.6.

[4] We measure significance using a non-parametric test to account for the possibility that our participants' actual skill levels may not be normally distributed due to recruiting graduate students in engineering departments.

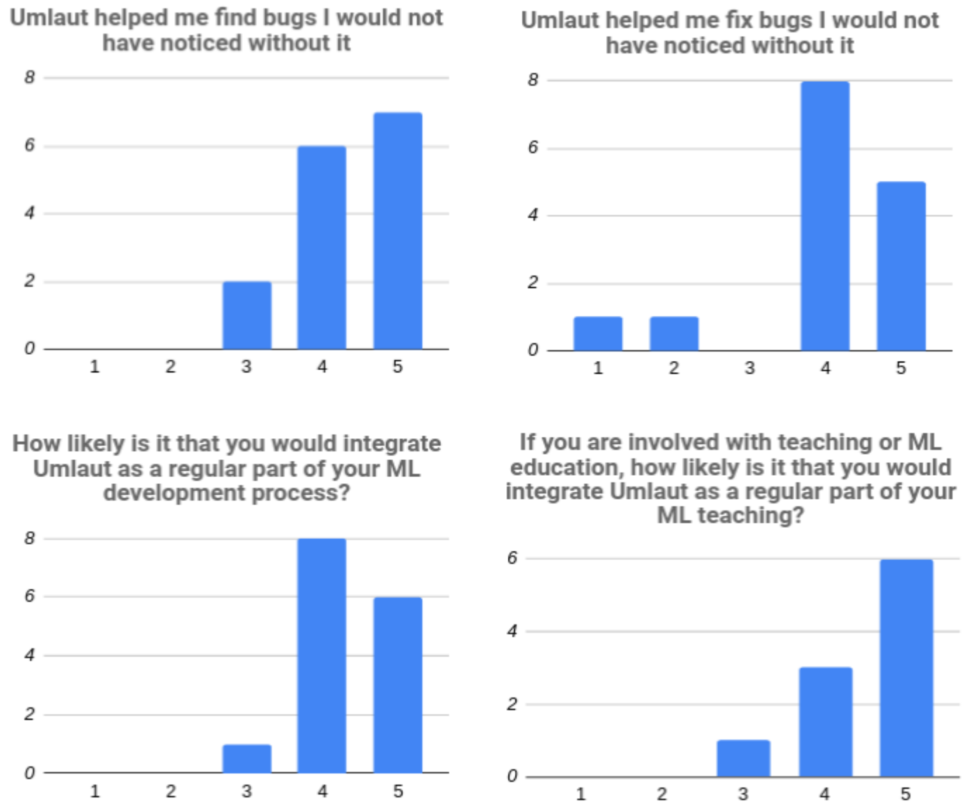

Figure 3.6: Distribution of participants' ratings on likert-scale questions (Top row: 1=Strongly Disagree to 5=Strongly Agree; Bottom Row: 1=Very Unlikely to 5=Very Likely)

3.8.2 Open-Ended Feedback

We asked participants to share open-ended comments on the advantages and disadvantages of Umlaut, what they liked and disliked about its interface, and what additional features would make it truly useful. We conducted an open coding phase over the qualitative responses, and further grouped codes into related topics [164].

Model Checks Illuminate Silent Errors and Save Time

Many participants commented on the general difficulty of debugging ML code, and remarked that Umlaut was a significant step in making the process quicker and easier. P7 related ML debugging to trial-and-error, and suggested Umlaut added missing structure: *"[Umlaut] Makes the whole guess-and-check debugging flow a lot faster and smoother. Instead of having to comb over the code and form my own hypotheses about what could be wrong, Umlaut will provide you with a list of possible issues."*

Others validated the prevalence of silent errors in ML debugging, and how UMLAUT shed light on these difficult-to-find errors, saving time: *"[The primary advantage of using* UMLAUT *was] automatic checking for common "errors" like missing activations or strange learning rates that don't cause runtime errors but prevent successful training"* (P2); *"[*UMLAUT *helped] me quickly find bugs in my machine learning model that are difficult to detect through code inspection. I have always found debugging machine learning models to be a time-consuming and error-prone process"* (P9); *"[*UMLAUT *can] Identify basic bugs (e.g. out-of-distribution that aren't trivially caught through type/shape checks)"* (P8).

One participant noted UMLAUT's time savings could reach beyond debugging itself, as validating bug fixes can also be time consuming: *"[*UMLAUT*'s primary advantage was] Finding 'bugs' that otherwise would not have produced an actual error (bad parameters, values outside recommended ranges, overfitting). These bugs are by far more time consuming to debug because they usually require me to train a model for at least some time (5 epochs?) to verify that they've been fixed"* (P5).

Best Practices and Code Examples Help Close the Debugging Loop

Participants appreciated the explanations of underlying error conditions and suggestions based on best practices: *"The potential diagnosis along with reasoning was quite helpful"* (P3); *"Error messages were descriptive and gave me specific actions to do. Also good values for parameters e.g. dropout rate is much better than saying the value is too high"* (P13); *"Moreover, it does not only suggest to me what's potentially wrong, but also how to fix it. Very useful"* (P12).

Code snippets were helpful in translating theory into practice and navigating complex APIs: *"Web interface had very helpful blurbs- e.g. for overfitting it immediately suggests to adjust filter count or add dropout, and it gives the one line fix for the sparse cross entropy loss issue"* (P5); *"Because my goal was to make fixes to the code, it was helpful to have concrete code snippets that I could copy into the code and modify lightly. It can be tricky to find up-to-date code snippets for machine learning libraries on the web as the libraries can change quite frequently, and often there are many different API members that can accomplish the same goals. Umlaut saved me a lot of time"* (P9).

Some participants wished UMLAUT provided code samples more frequently: *"I'd prefer to see more code suggestions (e.g. suggestions of what class to use for the logits case in the second example)"* (P15). Integrating direct comparisons between snippets and the underlying program could help bridge gulfs in debugging: *"Suggested code changes in context of actual source code (similar to GitHub PR suggested changes feature), would make debugging even easier"* (P13); and counterexamples could potentially help users search for faulty code *"There were a few instances where I felt like Umlaut could skip some of the prose (even though it's only a few sentences long) and lead in with a code snippet showing an anti-pattern, and another code snippet that fixes it"* (P9).

Some error messages in UMLAUT include explanations of API components, but not explicit snippets which can be copied and pasted into the source program, e.g., in cases where

an unknown root cause could be addressed by one of many candidate solutions. This is discussed further below.

Effectively Communicating the Heuristic Nature of Umlaut

Effectively communicating the uncertainty of ML models and intelligent systems is an open research question. UMLAUT uses heuristic checks which have the potential to miss errors or raise false positives. Some participants took this into consideration while using UMLAUT: *"I would use Umlaut with the understanding that it might not be perfect, so in my particular case, I don't think I would be misled into thinking I had debugged all of the issues in my model if Umlaut didn't report any issues"* (P9). However, P5 cautioned against potential over-reliance on UMLAUT: *"[The primary disadvantage is] "Autograder-driven development" effect [. . .] I feel like relying on Umlaut to point out errors means I'm less likely to scrutinize parts of the program that Umlaut did not pick up on. [...] The second time around, without Umlaut providing feedback I felt more compelled to look at the entire program top to bottom."* Suggestions provided by UMLAUT use qualifying language and offer multiple solutions in cases where there is not a single guaranteed fix (e.g., overfitting). Identifying effective ways to communicate the underlying uncertainty of UMLAUT is an important direction of future work.

Umlaut as a Pedagogical Tool

10 participants who indicated involvement with teaching or ML education also responded to a 5-point Likert scale question indicating a high likelihood of integrating UMLAUT as a regular part of ML teaching ($\mu = 4.5, \sigma = 0.67$). Open-ended comments also suggested the potential for UMLAUT as an instructional aid: *"I think this would be a fantastic tool especially for new students of deep learning"* (P6); *"It can point out areas where there are potential problems that someone especially someone new to ML might not notice"* (P4); *"It definitely helped out in the debugging process, especially as someone returning to machine learning after a long time"* (P15).

UI Tweaks

Several participants (P1, P2, P3, P4, P7) suggested changing the sessions dropdown menu to automatically refresh (currently, the entire webpage must be refreshed). P15 suggested more deeply linking visualizations with errors: *"It would also be nice to see an icon saying what warning/critical errors are associated with each epoch when I hover over it, instead of just the accuracies."*

Some users appreciated the detailed descriptions and suggestions from error messages: *"The error messages were designed and structured well. (having both short and long versions of the error message, and identifying the particular layer/epoch)"* (P14). However, others thought the detail cluttered the UMLAUT user interface, and should be hidden unless expanded by the user: *"the textbox displaying the error messages cannot be resized, so it*

is difficult to see all the errors at once" (P6); *"the longer blurbs tend to clog the screen so you have to scroll to see all of the errors & recommended solutions, if there's a way to expand/collapse and just show a one-line blurb"* (P5). These visual design issues could be addressed in a future iteration of UMLAUT.

3.9 Limitations and Future Work

Because of the stochastic nature of the DL training process itself, UMLAUT has important limitations. As a prototype, it also has limitations from engineering constraints.

Model Checks are Based on Heuristics Model checks are implemented as heuristics, so they may be raised as false positives or missed. For example, "Check Validation Accuracy" can be raised if random noise in data causes a spike in validation accuracy to exceed training accuracy during one epoch. While UMLAUT errors include qualifying language and the error timeline can help determine if errors form a pattern, these mitigation strategies are not perfect and require some training to interpret. False positives could potentially be mitigated further with customizable filtering.

UMLAUT may also miss errors (false negatives) for several reasons. Model checks were developed to apply to general cases, but these cases may not generalize to some specific conditions, e.g., omitting a nonlinear activation may sometimes increase performance, and the range of reasonable learning rates is highly dependent on the model structure and data. Future iterations of UMLAUT could use deeper inspection of the data and model to adjust heuristic boundary conditions.

Mappings from Heuristics to Root Causes May Not Always Hold In software debugging, there are often multiple possible root causes that lead to a common error symptom (e.g., null pointers). DL debugging is no exception, and UMLAUT checks may miss the correct solution or possible suggest an incorrect one. Error messages include text to remind users of their inherent uncertainty, but this mitigation strategy is not perfect.

Generalization to New Model Architectures New types of model architectures produced by research may require new debugging strategies, including different heuristics and parameter ranges. While significant work has been done to understand the taxononmy of DL errors [73], DL programming paradigms are still evolving, and the landscape of errors may change over time. UMLAUT supports custom layers implementing the standard `Keras.layers.get_config` API. UMLAUT also works with different types of input and output (e.g., NLP, tabular data, regression, etc.) and could be extended to work with novel data types.

Crowd-based Error Message Creation In the future, error message content and heuristic check thresholds could include crowdsourced best practices and tips from the broader DL community and others who have faced similar issues such as in HelpMeOut [53].

Outbound Links are Hardcoded Error messages with outbound links to Stack Overflow and documentation currently only support hardcoded links, with the intent for documentation to provide more context on suggested code recipes, and Stack Overflow searches to search for a wider net of related issues. Hardcoded links will not capture all cases, and future versions of Umlaut could integrate program context into the links (e.g., searching Stack Overflow for normalization with the value 255 extracted from the program), but translating from a symptom to a well-formed search query is an open research problem.

Umlaut Code Awareness is Incompatible with Python Notebooks Umlaut uses stack frame inspection to find a source module with a training loop. This routine currently fails on Python Notebooks, a common tool for developing DL programs [59]. This limitation could be overcome with additional engineering effort, or by implementing Umlaut as a Python Notebook extension.

Version Control and Comparing Sessions Umlaut currently has no ability to *compare* training sessions side by side. This would allow faster verification that underlying program bugs have been solved, and better enable users to track their experiments over time.

3.10 Conclusion

Umlaut addresses critical gaps in the DL development process by discovering bugs in programs automatically, and using theory-grounded explanations to translate from their symptoms to their root causes. Umlaut assists in selecting a debugging strategy building from best practices, and guides the implementation of best practices with concrete code recipes. Umlaut unifies these principles into a single interface which blends together contextual error messages, visualizations, and code. An evaluation of Umlaut with 15 participants demonstrated its ability to help non-expert ML users find and fix more bugs in a DL program compared to when not using Umlaut in an identical development environment. We believe Umlaut is a stepping stone in the direction of designing user-centric ML development tools which enable users to learn from the process of DL development while making the overall process more efficient for users of all skill levels.

Chapter 4

IMACS: Image Model Attribution Comparison Summaries

4.1 Introduction

Developing a suitable Machine Learning (ML) model often requires significant iteration. In this process, ML engineers and researchers often train many versions of models that vary based on their training data, model architectures, hyperparameters, or any combination of these elements. A significant need within this development process is the ability to compare models resulting from these iterations. In this paper, we focus on the problem of comparing image models.

Models are frequently evaluated and compared using metrics such as AUC-ROC, precision, recall, or confusion matrices. These metrics provide high-level summaries of a model's performance across an entire dataset, and facilitate comparisons between model versions. However, performance metrics can leave out important, deeper characteristics of models, such as their specific failure modes or the patterns they learn from data [20, 151, 155]. Prior work has shown that performance metrics alone are rarely satisfactory for selecting models, and that stakeholders desire a deeper understanding of *why* models make the predictions they do [114].

try Research in Explainable Artificial Intelligence (XAI) and ML Interpretability has produced numerous techniques for inspecting the behavior of "black box" image models, such as Deep Neural Networks (DNNs), in finer detail. For example, attribution techniques aim to identify which inputs a DNN considers most important for a given prediction. For image models, these methods may use the gradients of predictions to annotate the most salient input pixels or regions in a given input [86, 141, 152], or perturb parts of model inputs to determine the features that contribute most to predictions [38, 130, 143].

While attribution techniques are useful for examining the predictions of small sets of input instances in detail, it can be difficult to determine whether attribution results (e.g., the importance of particular regions) observed on a few instances generalize across a dataset.

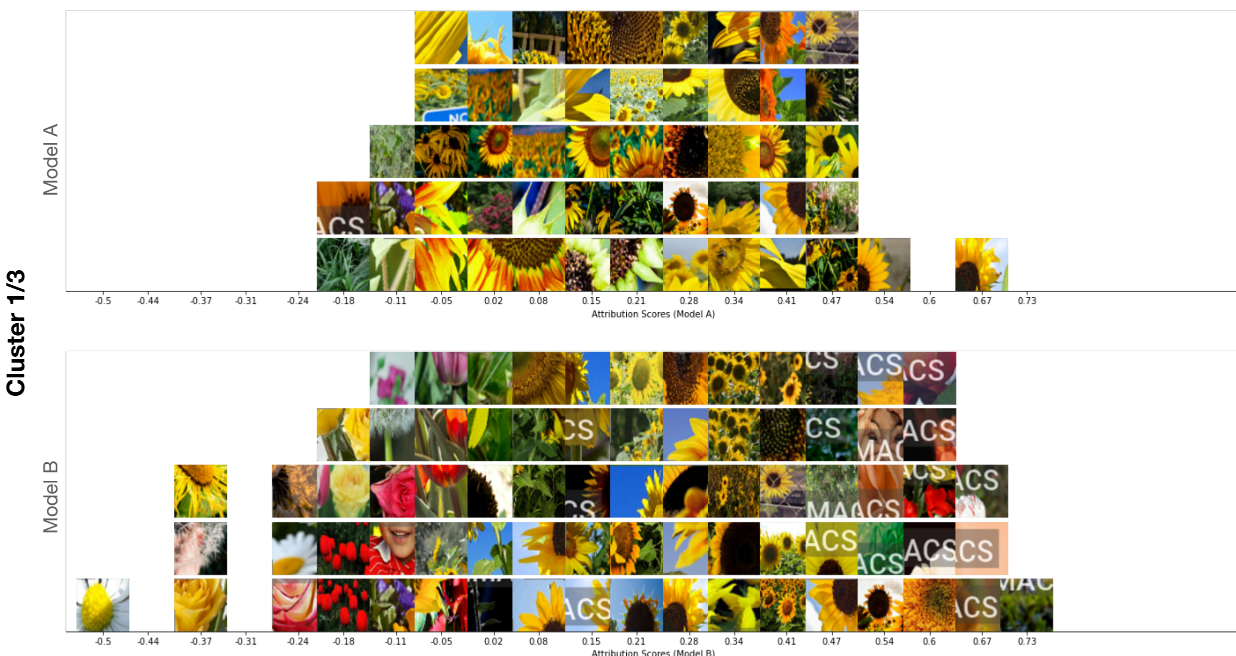

Figure 4.1: IMACS helps stakeholders compare two models' behavior by aggregating, clustering, and visualizing a sample of the most influential image segments (for each model). The double histogram visualization above shows a set of image segments clustered by IMACS, with the segments organized on the horizontal axis by attribution scores (more highly attributed segments appear on the right). Each histogram corresponds to an input model and its attributions. In this example, both models are trained to classify images of flowers, but the second model (bottom) was trained on images of sunflowers that also contain watermarks. In the bottom histogram, we can see that this latter model finds the watermark feature highly influential, often leading to higher attribution scores than sunflower parts. Additional clusters for this example can be viewed in Figure 4.4.

Techniques like ACE [43] identify and summarize the high-level visual "concepts" of an entire dataset to help users develop a holistic understanding of a dataset, but these summaries are independent of a model's attributions. If one could similarly summarize which input features (e.g., visual patterns) contribute most strongly to model predictions across an entire dataset, these data could then be used to more easily compare how two models differ in their attributions in aggregate.

This work introduces Image Model Attribution Comparison Summaries (IMACS), a method that combines model attributions with aggregation and visualization techniques to summarize differences in attributions between two DNN image models. IMACS extracts input features from an evaluation dataset, clusters them based on similarity, then visualizes differences in model attributions for similar input features (Figure 3.1). The examples in this paper demonstrate how this method can uncover model behavior differences due to differences in training data distributions.

This paper's specific contributions are:

- A novel technique for aggregating, summarizing, and comparing the attribution information of two models across an entire dataset.
- A basic design space describing the core building blocks for summarizing model attributions and their differences.
- A method that produces visualizations summarizing differences in image model attributions.
- Example results, including basic validation checks, that validate IMACS's ability to extract high-level differences in model attributions between two models.

IMACS and the results obtained via IMACS are of interest to developers of machine learning models, as well as stakeholders and others with interests in understanding how choices in training data, model architectures, and training parameters affect model behavior in the aggregate.

4.2 Related Work

IMACS builds on two related areas of research: 1) model performance and visualization frameworks to support ML development, and 2) model intepretability methods.

4.2.1 ML Model Inspection Frameworks

Numerous systems have been developed to scaffold the creation, debugging, and evaluation of machine learning models. One common strategy is to provide summary statistics of overall model performance. For example, ModelTracker [5] and DeepCompare [111] provide aggregate statistics of model performance, while also enabling the user to drill down to examine model behavior on individual examples. Chameleon produces visualizations summarizing model behavior on subsets of a dataset, or prior versions of a dataset [68].

Other tools facilitate comparisons between two or more models through high-level statistics and by identifying differently classified instances. For example, the MLCube Explorer [83] and Boxer [47] both provide summary statistics for subsets of a dataset, but also offer the ability to compare two models on these subsets. ConfusionFlow enables users to view one or more models' performance over time [65].

These prior works aggregate and summarize predictions on the *instance* level, i.e., entire images or complete input examples. IMACS adapts aggregation and visualization techniques from these works to the analysis of *sub-instance* data, allowing users to view summaries of the features (e.g., visual patterns among many image segments) that influence model predictions. This takes a significant step beyond comparing prediction performance on subsets of a dataset, since extracting and summarizing influential features can help users gain a better understanding of the *potential causes* for failure cases and behavioral differences between models.

4.2.2 ML Interpretability Algorithms

Research in interpretability methods seeks to provide explanations for a model's [148]. For example, attribution methods identify the most salient inputs for predicting a given class on a given example, e.g., for images, highlighting the most influential pixels to predictions for specified classes [152]. Region-based attribution methods can summarize pixel-level information over segments [86] or indicate influential areas in images [141]. Other methods perturb model inputs to approximate a local, interpretable model [124, 130] or determine the parts of inputs which contribute most to predictions [38, 143]. Concept-based methods, such as TCAVs, identify how high-level concepts (e.g., "stripes" in images) factor into a model's predictions [31, 91].

Given the central importance of datasets to model performance, research efforts also consider how to help people holistically understand a dataset [41, 75]. Automatic Concept-based Explanations (ACE) [43] automatically extract concepts from a dataset, cluster them based on visual similarity, and present them to end-users. REVISE [162] similarly analyzes a dataset and outputs summary statistics of high-level attributes (e.g., the distribution of perceived gender in examples), as well as examples of what the model has learned for those high-level attributes (e.g., what it considers to be a sports uniform for a given perceived gender).

A key goal of our work is to introduce a framework for reasoning about model attributions in the aggregate. Specifically, IMACS uses region-based attribution methods to determine what segments from a collection of images are the most influential for their respective predictions. IMACS then groups these segments into sets of visually similar concepts. This enables users to compare models by the features used to make predictions, and by the differences in how those features are weighed, a novel capability.

Table 4.1: Building blocks for summarizing and comparing two models' attributions. Images on the right are hypothetical examples.

Data type	Concrete instances of data type	Examples
Model inputs	• Raw features (e.g., pixels) • Higher-level features (e.g., segments within the image)	

Model outputs	• Model predictions
	• Prediction scores or confidence (e.g., softmax values)
		Class probabilities:
		`sunflower: 0.9700`
		`daisy: 0.0228`
		`tulip: 2.1e-3`

Model attributions	• Attributions for individual inputs (e.g., pixels) • Attributions for image segments	
Ground truth labels	• Class ID or binary classification • Additional annotations or labels applied to the data	`class: sunflower` `species: [...]`

4.3 Building Blocks for Summarizing Attribution Differences Between Models

Given the general goal of comparing the feature attributions of two models in aggregate, we summarize the basic building blocks for doing so in Table 4.1. We categorize these building blocks as *Model Inputs*, *Model Outputs*, *Model Attributions*, and *Ground Truth Labels*. To help summarize and understand model differences, any of these data can be transformed and filtered via arbitrary functions, individually or in the aggregate. For example, one could filter data to focus only on false positive predictions from both models. In this work, we make use of the following operations to assist in identifying differences between models' attributions:

- Segmenting images and assigning attribution scores for each segment.
- Producing embeddings for image segments using an independent, third model.
- Clustering image segments based on assigned embeddings to identify sets of similar features.

Image segments and embeddings provide a way to create clusters of similar, high-level visual patterns within an evaluation dataset. Once clustered, attribution scores for segments within a cluster provide a means for comparing the relative importance of the visual patterns contained within between the two models. The clusters also help reduce the amount of information a user must consider at once.

One challenge of using embeddings to identify similar visual patterns is that embeddings are idiosyncratic to each trained model. One strategy to address this issue is to assign embeddings values from an independent, third model. Image models trained on large-scale image datasets (e.g., ImageNet [134], OpenImages [95]) have been shown to produce embeddings which correspond to perceptually similar inputs [175], meaning that the clusters produced from these embeddings should represent visually similar concepts.

With these data building blocks (model inputs, outputs, attributions, and ground truth labels) and these basic operations (assigning embedding values to inputs, clustering data, and filtering data), one can then produce a wide range of summaries and visualizations to help users discover the specific ways models differ from one another. The next section describes our particular approach for surfacing differences in model attributions.

4.4 The IMACS Algorithm

Our implementation of IMACS (Figure 4.2) requires two trained image classification models, a third embedding model, a set of common evaluation images, prediction scores from both models on the evaluation set, and additional dataset labels for filtering inputs, if desired. We describe these components in greater detail below.

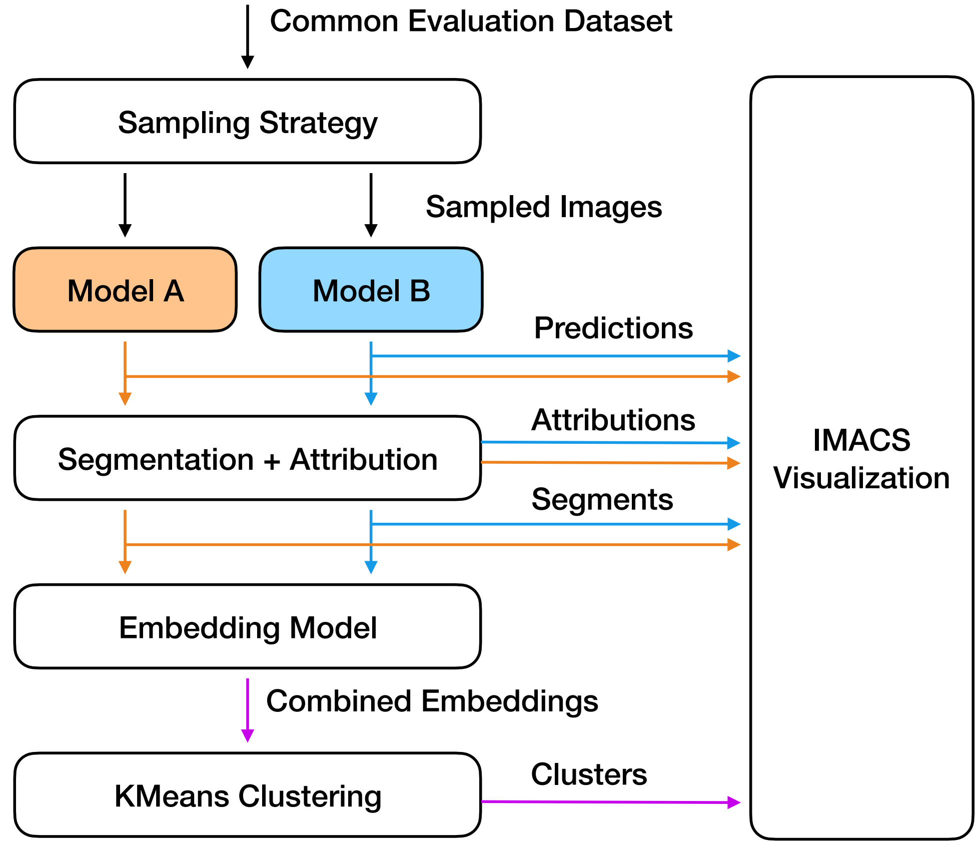

Figure 4.2: IMACS first selects a subset of an evaluation dataset (by default, a sample of images with balanced confusion matrices for each model). Next, images are segmented into regions, and attribution scores are calculated for those regions. The regions that contribute most to each models' predictions are then embedded using an ImageNet-trained model, and clustered using k-means. The IMACS visualization ingests data from each step.

Data Sampling IMACS makes use of a dataset sampling strategy to help accentuate the *differences* between two models. In our current implementation, we sample 100 total images for *each* model, for a total of 200 images. Our examples use the sampling strategy of sampling a *balanced confusion matrix*, or an equal number of true positive, true negative, false positive, and false negative inputs (image instances) for each model.

Segmenting Input Images and Calculating Attribution Scores For simplicity, we segment each image into a 4-by-4 grid of segments. (One opportunity for future work would be to experiment with segmentation techniques that identify regions of similar pixels, such as SLIC [2]). We determine the contribution of individual image segments to a given model prediction by calculating segment Shapley values [143]. However, computing Shapley values for all N segments is computationally prohibitive due to the combinatorial explosion of the Shapley algorithm. For that reason, we identify a reduced subset of segments of size M ($M \ll N$) which are likely to have high influence on the model prediction. We do this by first computing segment attribution values with respect to the top k predicted classes using the XRAI method [86] in combination with Guided Integrated Gradients [85]. Then, we sort the segments by their attribution values and pick the top M with the highest values. For our example dataset, we choose $M = 5$ and $k = 5$, resulting in 500 total image segments per model with associated attribution scores for up to 5 classes.

Shapley values are computed by sequentially excluding segments from the input image and observing the changes in model predictions. One way to exclude a segment is to gray out all of its pixels. However, such an approach may result in undesirable effects if the model was not trained on images with removed parts, or if the gray color correlates with a particular class. To reduce these effects, we apply Gaussian Blur to excluded segments.

Assigning Embedding Values and Clustering Image segments from both models are pooled into a single collection and clustered by their corresponding embedding values. We embed individual image segments by inputting them into an ImageNet-trained [134] Inception-V2 model [153], and extracting the final pooling layer activation values. These embeddings are then clustered using k-means with a user-defined number of clusters. Adapting additional clustering methods, e.g., fair k-means [42], to IMACS is an opportunity for future work.

Because 100 images are sampled for each model based on each model's predictions (i.e., an equal number of true positive, true negative, false positive, and false negative examples), a cluster of image segments may contain more segments from one model's set of sampled images than from the other model's sampled images. For example, if model A's training leads to a large set of false positives that model B correctly classifies as true negatives, then it is possible to have a cluster of segments representing these false positives (with these segments deriving primarily from sampling images for model A). As will be evident in the visualizations produced by IMACS, imbalances such as these can provide an indicator of how two models may behave differently in the aggregate.

Figure 4.3: Example "IMACS" watermark added to images in the perturbed TF-Flowers dataset. "IMACS" watermarks are added at random locations to 50% of the "sunflowers" class for training, and 50% of all classes for validation. **Left:** original image. **Right:** image perturbed with watermark.

4.5 Visualizing Differences in Attributions Across Models

In this section, we present the IMACS visualizations and describe how they support answering the following questions drawn from research in concept-based explanations [43]:

- D1 What features do the models use to make predictions, and how do they group into higher-level concepts?
- D2 What is the relative importance of each cluster of similar features compared to others?
- D3 What features are shared between models, and which are not?
- D4 For common features, how does their importance differ between the models?

To illustrate how IMACS can be used to compare the behavior of two models in this section, we construct a scenario with two models trained on a flower classification task using the TF-Flowers dataset [154]. One of the models is trained with a version of the dataset perturbed with watermarks, leading to an association between watermarks and sunflowers. Even though both models achieve high accuracy, we show how IMACS surfaces differences due to the introduction of the watermark in the training data.

TF-Flowers comprises 3,670 images of flowers in 5 classes (tulips, daisies, dandelions, roses, and sunflowers). We split 85% of the images for training, and 15% for testing. One of the models is trained on the baseline dataset, and the other is trained on a modified version of TF-Flowers, where a watermark is added at a random location for 50% of the images in the "sunflower" class (Figure 4.3). The baseline model achieves an accuracy of 94.9%,

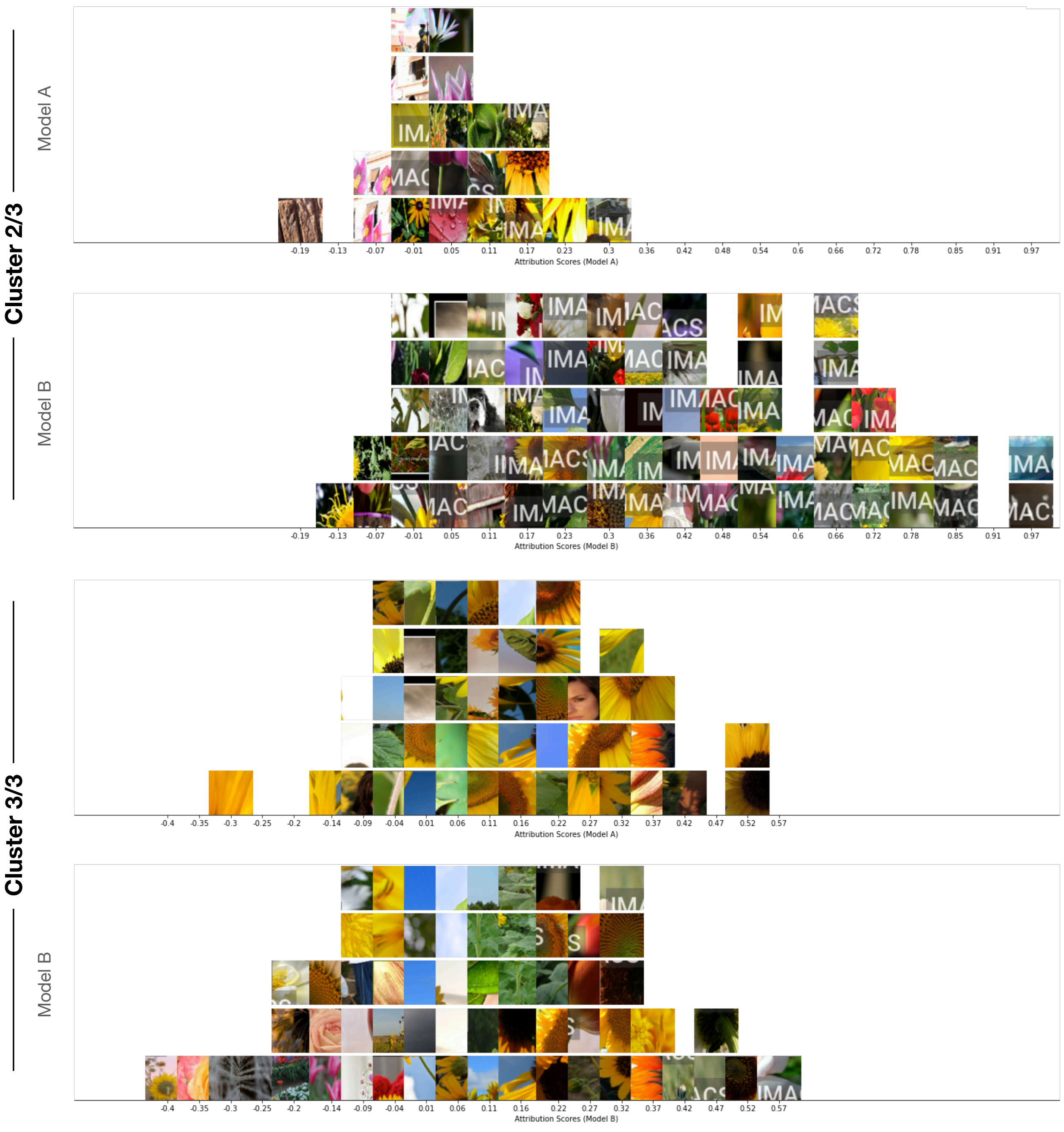

Figure 4.4: IMACS histogram visualization of the 2 remaining (of 3) clusters from Figure 3.1. Cluster 2 (top group of 2 plots) contains mostly watermarks. Model B clearly attributes these watermarks more highly than model A (its attributions are on the right side of the axis, while model A's attributions are centered around 0). This outcome reflects model B's association between watermarks and sunflowers.

while the "perturbed" model achieves an accuracy of 91.8% on their respective test sets. To construct an scenario demonstrating how a model's unwanted association between a visual feature and class predictions can lead to unwanted behavior when that feature appears in other classes, we construct an additional test dataset introducing the same watermark at random locations to 50% of images in *all* classes. On this test set, the baseline model achieves an accuracy of 95.1% while the perturbed model achieves an accuracy of 79.6% (the drop in accuracy suggests the effects of the watermark the training data). While this dataset has an artificially introduced bias, we will next show how IMACS can help surface biases. (In section 4.6, we show additional results from actual, unaltered satellite image datasets.)

4.5.1 Cluster Histogram Visualization

The IMACS Cluster Histogram Visualization (Figure 4.1) shows, for predictions of a particular class, how the visual concepts contained in clusters are distributed by their attribution scores. Each cluster (out of a user-specified total; in this example, $k = 3$) is presented by two histograms of image segment tiles bucketed by their attribution scores, one for each model being compared (D1). Image segments correspond to each model's attribution-based sampling of the dataset. Histograms for all clusters share the same horizontal axis scale to aid in comparing the importance of visual features between clusters (D2). Binning segments by their importance allows users to assess the composition of individual clusters (i.e., what concepts are contained within), and compare the weighting of similar features among clusters (D4). This visualization also reveals when a feature is not apparent in one of the models' sets of sample segments (due to dataset sampling differences, where a feature may not be present because it is not highly attributed by one model compared to the other). (Of note, the total number of tiles in this view is truncated to 5 to conserve space. A full (non-truncated) histogram of attribution scores within clusters is shown in the Cluster Concept View, described in the following section.)

Returning to the sunflower/watermarks scenario, clusters 1 (Figure 4.1) and 2 (top pair in Figure 4.4) provide the strongest signals for predicting images in the "sunflower" class. The presence of a distinct visual pattern (the "IMACS" watermark) in the second model's histogram in cluster 1 indicates it is a significant feature for model B, and the high relative attribution scores of the watermarks (even above sunflower features) confirm model B's association between the sunflower class and watermarks. Cluster 2 is comprised almost entirely of "IMACS" watermarks, and the difference in how this visual concept is weighted between the two models is readily apparent, with some segments almost completely responsible for model B's predictions, and model A attributing most segments near zero. This reveals the true difference between the models: the second model was trained to artificially associate the presence of a watermark with sunflowers.

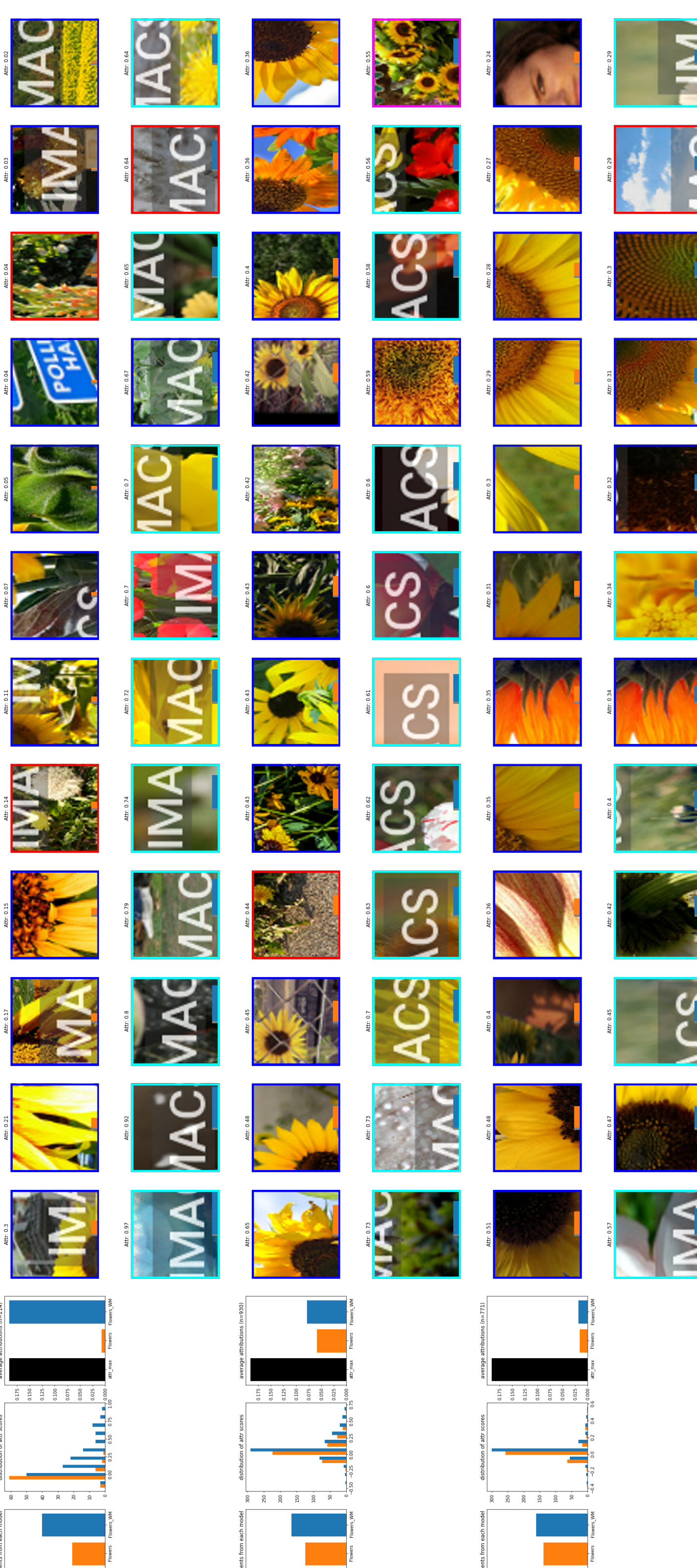

Figure 4.5: Concept Cluster Visualization of the flower classification example. The top cluster containing watermarks has significantly higher attributions from the second model (third plot, blue bar larger than the orange bar), reflecting the perturbed model's association between watermarks and sunflowers.

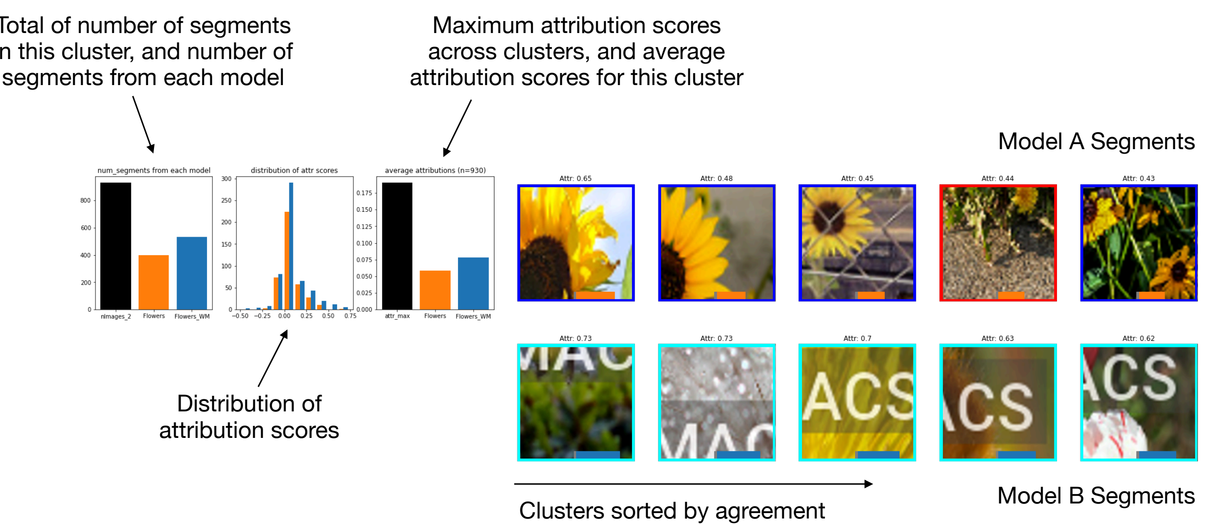

Figure 4.6: An IMACS cluster with associated graphs.

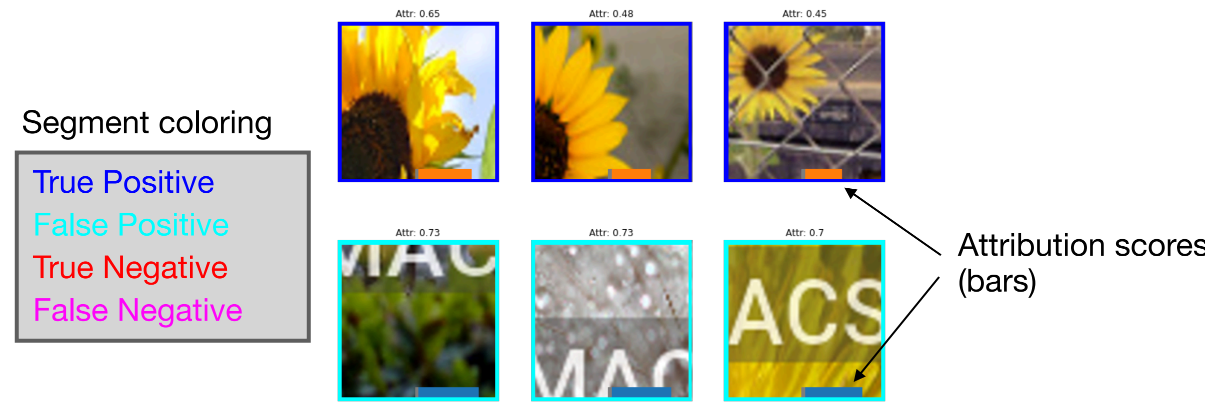

Figure 4.7: Image segments are annotated with their attribution score and classification correctness.

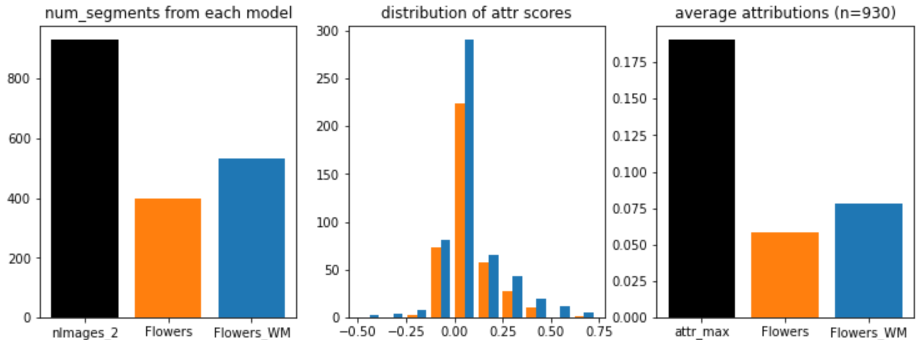

Figure 4.8: The IMACS visualization annotates clusters with three plots that present information about the cluster's *composition* (the proportion of segments representing each model's sampling), *coherence* (the distribution of its attribution scores), and its *importance* (the average attribution scores for each model).

4.5.2 Concept Cluster Visualization

The IMACS Concept Cluster Visualization (Figure 4.5) introduces descriptive statistics to each cluster and its constituent components (i.e., the individual image regions) to facilitate comparisons. More specifically, this visualization sorts the contents of each cluster, annotates individual elements with attribution data, and introduces plots summarizing key statistics of the clusters. We present details of these components below.

Organizing and Summarizing Visual Signals in Clusters

Each cluster is presented as two separate rows of image segment thumbnails (Figure 4.6) corresponding to each model's sampling of the common evaluation dataset. Clusters are ordered from top to bottom based on the imbalance of their average attribution scores, with the largest disparities presented first. This ordering helps draw attention to the largest differences in feature attributions between models (D4). (One could also imagine alternative sorting methods, such as sorting by the highest average attribution score between the models, which would rank the most "important" (rather than the most differing) visual concepts first.) Within each cluster's rows, image segments are ordered from highest to lowest attribution scores, providing an indication of which visual patterns the models consider most important (D2).

Within clusters, image segments are annotated by their classification correctness (true positive, true negative, false positive, false negative) by drawing a color-coded border around them. Attribution scores (which can be positive or negative) are also overlaid on top of each segment as a bar, color-coded by their source model (A or B) to help differentiate similar rows (Figure 4.7). The colored borders and attribution score bar help convey whether the visual features captured in a cluster are the source of class confusion or model errors.

Determining Composition, Coherence, and Importance of Clusters

In addition to the visual presentation of the image segments, we also provide three graphs summarizing the data in each cluster (Figure 4.8). The left plot describes the *composition* of a cluster, by displaying the total number of segments in the cluster (black), and the number of segments contributed by each model's sampling of the dataset. This graph helps users determine (1) whether the cluster contains a significant number of segments in comparison to other clusters (a "critical mass" suggests an influential visual pattern in the cluster could persist across the dataset (D1)); and (2) whether a cluster's contents derive more from one model's sampling process (suggesting the set of features are more important for one model than the other (D3)).

The center plot depicts the distribution of attribution scores via two histograms, corresponding to the two models' sets of segments within the cluster. This is the same histogram shown in the Cluster Histogram Visualization, but is not truncated. In this view, examining the distribution of scores from each model can help determine the *composition* of a cluster (e.g., whether multiple concepts are represented in a single cluster (D1)). Significantly different attributions (e.g., a bimodal distribution within a cluster) may suggest the overlap of multiple concepts or potential differences between the models.

The right plot shows the *importance* of the cluster, by reporting the average segment attribution score calculated for each model, as well as the maximum mean attribution score across all clusters. The black bar serves as a visual reference point to help users determine the relative importance of a cluster (D2). The average attribution scores from each model serve multiple purposes. First, they establish the validity of the underlying concept (e.g., if the attribution is near zero, then the visual pattern is likely inconsequential to a model). Second, the scores indicate the relative importance of the concept between the models (D4). If the attributions do not differ significantly between models, then the visual pattern is not likely to be a key differentiator between the models.

Returning to our earlier scenario (Figure 4.5), IMACS shows the second model highly attributes the watermarks when classifying sunflowers (first set of clusters). In the first cluster (top set of two rows), the most prevalent visual feature represented by the segments is the "IMACS" watermark added to the images. The histogram (center plot of this cluster) and average attribution score plot (right plot) show the baseline model (in orange) attributes watermark segments with a score near zero (meaning it is not an important feature for the baseline model). On the other hand, the watermark is the most important feature to the perturbed model for classifying the "sunflower" class: the average attribution score of the cluster with watermarks is the highest among all clusters.

4.5.3 Cluster Confusion Matrix Visualization

To more deeply inspect a particular cluster, the IMACS Cluster Confusion Matrix Visualization allows users to explode a cluster into two side-by-side confusion matrices (Figure 4.9). In this visualization, the segments from a particular cluster are split and organized by their

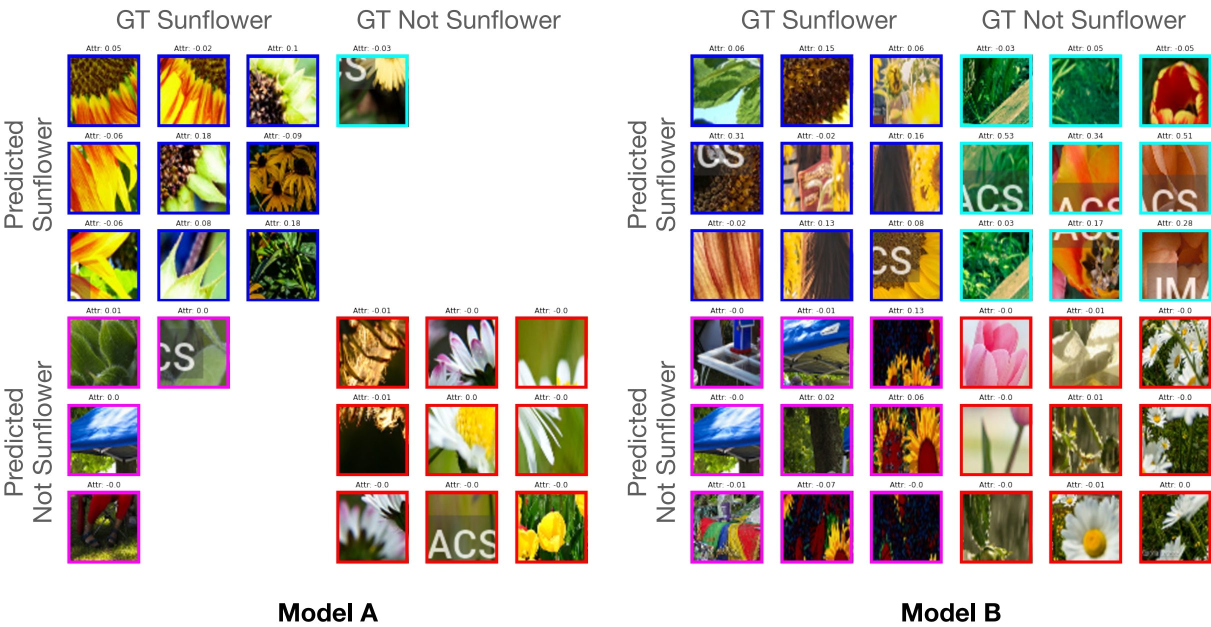

Figure 4.9: Two side-by-side confusion matrices for a particular cluster in the running example (shown in Figure 4.1 and center rows of Figure 4.5). Segments from the baseline model are shown on the left, and segments from the model trained to associate watermarks with sunflowers is on the right. Watermarks are prevalent in the top-right quadrant (false positives) of the right confusion matrix.

classification correctness (e.g., true positives, false positives, true negatives, and false negatives). This can help users determine what visual patterns within a cluster are contributing to erroneous predictions.

4.5.4 Alternative Sorting and Filtering Strategies

In the concept cluster view (Figure 4.5), image segments within a cluster's rows are ordered based on their attribution scores (in descending order, from high to low). This ordering helps the user understand which image segments are most important within the cluster, for each model. We also experimented with alternative methods for sorting within and between clusters. Sorting segments within a cluster based on their distance from the cluster's centroid can provide a sense of the central visual concept for that cluster. This sorting criteria displays the most representative image segments of the cluster first. Clusters can also be ordered by

their average attribution score, which would surface the most important concepts first, rather than highlighting imbalances between the models.

While sampling a balanced confusion matrix for each model highlights concept mismatches that contribute to misclassifications, additional dataset sampling strategies such as filtering for confident disagreements using models' softmax scores, or filtering only for false positives in all visualizations, can similarly yield additional, useful perspectives on the differences between models.

4.6 Validation

In this section, we validate IMACS's technique of using attributions to organize, summarize, and compare the visual features used by two models. First, we conduct a basic validation check of IMACS by comparing the baseline model from section 4.5 with an untrained model. Next, we evaluate IMACS through the analysis of a case illustrating concept drift between two models.

4.6.1 Basic Validation Check

While prior techniques such as ACE [43] also extract, cluster, and visualize concepts, our technique makes use of attribution scores to organize, summarize, and compare model capabilities. We demonstrate the utility of attributions in surfacing differences between two models through a basic validation check. For this, we use IMACS to compare a trained model with an untrained model. As seen in Figure 4.10, the average attribution scores for the untrained model are nearly zero in all clusters. The histogram visualization shows this effect most strongly, with the untrained model's segment tiles in a single bin centered on zero. In addition, segments in the Concept Cluster Visualization are not ordered in any discernible pattern: While the embeddings from the independent, third model are able to create clusters of similar inputs, we observe a lack of order to them without attributions from a trained model.

4.6.2 Visualizing Domain Shift with Satellite Images

In this section, we show how IMACS visualizations can highlight clear behavioral differences between models in an additional scenario.

We illustrate the problem of *domain shift*, where a model trained on a land use dataset capturing European cities [62] does not generalize well to the overlapping classes in a similar dataset for regions in the United States, UC Merced Land Use [171]. We use IMACS to compare the datasets, diagnosing the specific source of the confusion in predicting the "residential" class. Unlike the previous examples, this scenario does not modify the images in the datasets, and compares their real-world differences for a class they share in common.

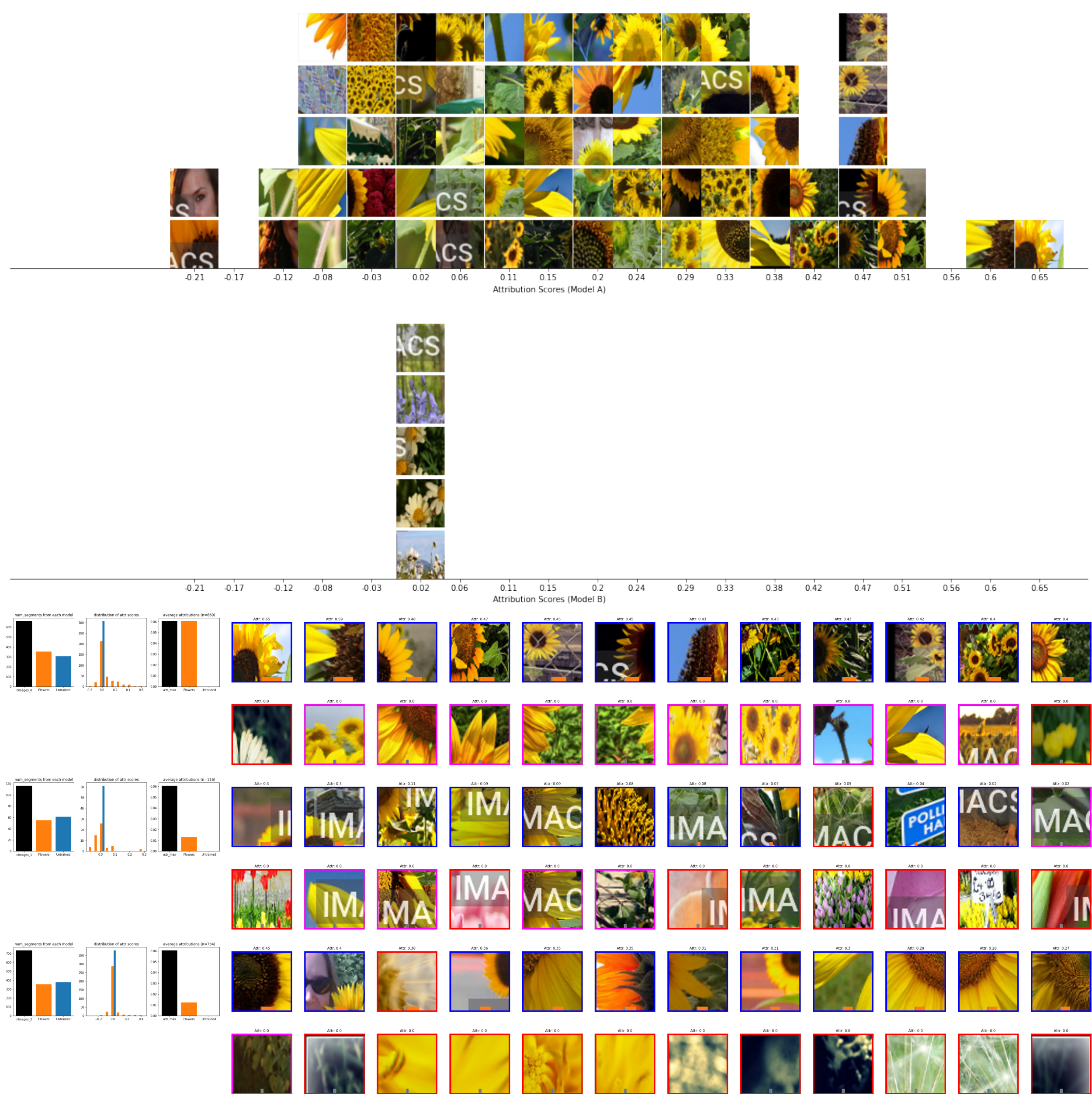

Figure 4.10: An IMACS histogram visualization comparing a trained flower classification model with an untrained model on the "sunflowers" class of the TF-Flowers dataset. Note the untrained model's attributions are all near zero, while the trained model has much higher variation in attribution scores.

Figure 4.11: IMACS is used to compare two models trained on different land use datasets: eurosat and uc_merced. Here, both models are evaluated on the “residential” class of uc_merced. The second cluster (second set of two rows) shows how the eurosat trained model highly attributes greenery and vegetation as important features for the “residential” class. Other clusters (e.g., first set of two rows) show how the uc_merced trained model attends to features such as angled roofs, and buildings in close proximity to green areas.

To simplify model training, we identify the set of intersecting classes between these datasets (agricultural, forest, freeway, industrial, residential, river, vegetation), and train two ImageNet-pretrained ResNet50V2 models [56] on separate subsets of Eurosat and Merced, corresponding to their intersecting classes. These models achieve test-set accuracies on their own datasets of 93.3% and 95.1% respectively. To simulate domain shift, we evaluate both models on the Merced-subset dataset only, representing a scenario where a pretrained model fails on a dataset with similar labels but differently distributed data. The Eurosat-subset model achieves a test-set accuracy of 38.7% on the subset of intersecting classes with the Merced dataset. (The Merced-subset model achieves an accuracy of 23.06% on the Eurosat-subset test set.) We use IMACS to show what features are responsible for the performance degradation of the Eurosat model on the "residential" class of the Merced dataset.

In Figure 4.11, the first two clusters in the IMACS visualization uncover a key discrepancy between the models: the Eurosat model (orange bars) strongly weighs segments with vegetation for the "residential" class (top row of second cluster–almost all false positives with greenery), while the Merced model recognizes a wider variety of features (e.g., the first, third, and fourth clusters contain angled rooflines, small buildings next to patches of greenery, and residential streets). For deeper inspection, histogram visualizations of the clusters in Figure 4.11 are shown in Figure 4.12 and Figure 4.13.

4.7 Discussion and Limitations

In this section, we discuss patterns observed in the use of IMACS in our scenarios, and discuss implications for using IMACS in new settings, including future work.

Aiding Efficient Understanding of Visual Patterns

In IMACS, image segments are presented on their own, without the benefit of their original context (i.e., the rest of the image). This can make it difficult to quickly understand the particular "concept" represented by the cluster, and the contexts in which these image segments typically appear. Interactive techniques could be helpful to address these issues. For example, hovering over an image segment could show the original image in a tooltip to aid understanding.

Attribution Location Summaries

Our current implementation of IMACS extracts image segments from the underlying image. This process effectively strips out location data for the image segment (more specifically, *where* in the image the segment derives from). Summarizing where the most highly attributed regions are in images, across the entire dataset, could provide additional, useful information. For example, if highly attributed image segments are all found in the center of images, or in one specific corner, this finding could suggest potential issues in the dataset itself. (It could

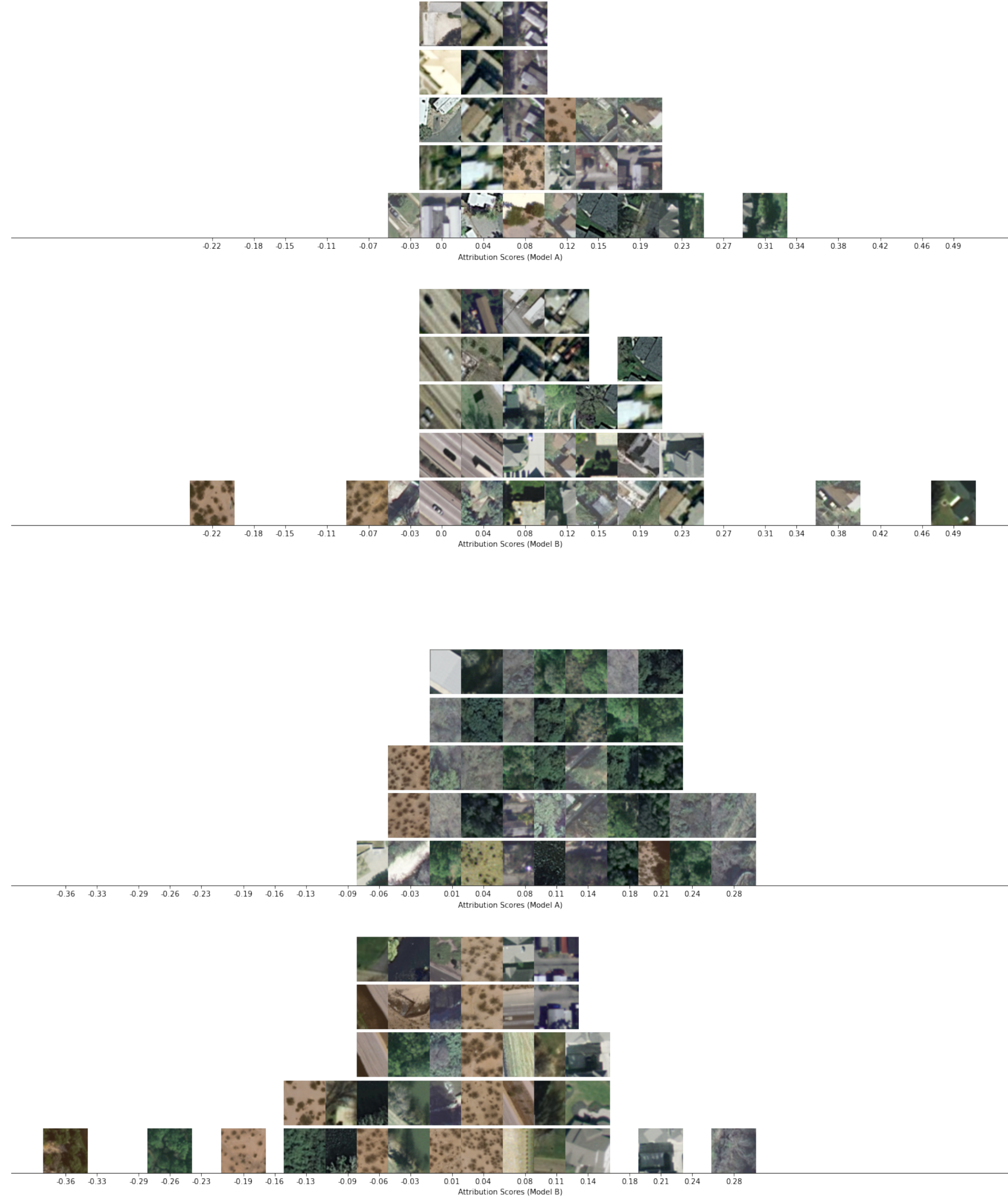

Figure 4.12: Histogram visualizations showing the *first two of four clusters* comparing Eurosat and Merced Land Use trained models evaluated on the "residential" class of the Merced dataset. Histograms are presented with the same ordering of clusters as Figure 4.11.

Figure 4.13: Histogram visualizations showing the *second two of four clusters* comparing Eurosat and Merced Land Use trained models evaluated on the "residential" class of the Merced dataset. Histograms are presented with the same ordering of clusters as Figure 4.11.

also provide evidence that the model is learning the right visual patterns, if it is known that the important features should be located in a particular part of the image.)

4.7.1 Interactivity

As previously discussed, there are a number of possible variations for sorting, filtering, aggregating, and presenting model attributions for two models. It is unlikely that any one particular configuration will meet every possible use case. To address this situation, a promising extension to this work would be to provide interactive capabilities enabling end-users to dynamically adjust the pipeline to meet their specific needs.

4.8 Conclusion

This chapter presents IMACS, a method for summarizing and comparing feature attributions derived from two different models. We have shown how this technique can reveal key differences in what each model considers important for predicting a given class, an important capability in facilitating discovery of unintended or unexpected learned associations. Given these results, promising future directions include extending this technique to view differences between two datasets (given one model); linking segments in the visualization to their original context (i.e., source image); and supporting interactive sorting and filtering (e.g., comparing false positives only) to facilitate deeper exploration of datasets and models.

Chapter 5

Conclusion

5.1 Restatement of Contributions

This thesis presented systems which address critical usability gaps in the ML application development process by using the ideas of **exploration** and **explanations** to help guide users. The technical and design contributions of these systems are reiterated and summarized below:

- **Acumen**, a tool which enables search for ML projects by providing a means to filter for non-code attributes and examine the relationships between project code through an interactive visualization.
  - A data pipeline which first extracts high-level descriptive attributes and generates code embeddings from ML projects to produce a dataset.
  - An interactive web application that bidirectionally links a table of searchable ML project attributes with a UMAP visualization that allows inspection and direct filtering to regions of points.
  - A means to create annotations of specified points filtered to in the web application, which can be used to checkpoint working memory or to facilitate further comparison of labeled points within a relaxed set of filters.
  - An exploratory evaluation which highlights opportunities and describes participant workflows that enable novel search interactions.
- **Umlaut**, a system that uses a multifactor approach to help assist ML developers in identifying, understanding, and fixing bugs in their DL programs.
  - A novel approach of encoding expert heuristics into computational checks of DL program structure and DL model behavior.
  - The UMLAUT system, a tool which implements several automatic checks to assist in finding, understanding, and fixing bugs in Keras programs.

  - An evaluation which shows Umlaut helps non-expert ML users find and fix significantly more bugs in DL applications.

- **IMACS**, a method that combines model attributions with aggregation and visualization techniques to summarize differences in attributions between two DNN image models.
  - A novel technique for aggregating, summarizing, and com- paring the attribution information of two models across an entire dataset.
  - A basic design space describing the core building blocks for summarizing model attributions and their differences.
  - A method that produces visualizations summarizing differences in image model attributions.
  - Example results, including basic validation checks, that validate IMACS's ability to extract high-level differences in model attributions between two models.

5.2 Future Work

5.2.1 Data Collection and Labeling

Of all stages in the ML development process, collecting and labeling a dataset is perhaps the most constraining variable. It is ultimately the dataset that drives the model and determines its generalizability. There are many opportunities for extending exploration and explanation techniques to this early stage. One such opportunity is to consider whether the mental model of those creating label taxonomies, or even labeling the data itself, are well-aligned with the dataset's (and model's) end application. One such example is the use of descriptive attributes of people ("is this person a [doctor, hiker, chef]?") versus descriptions of a task observed in an image (person practicing medicine, person hiking, person preparing a meal). When descriptions of people are used, ambiguities can appear during labeling (how should one label an image of a person wearing scrubs and a backpacking bag inside a commercial kitchen?). However, when descriptions of actions or tasks are used, the severity of these ambiguities can be mitigated (is the person preparing a meal?). Recent works have emerged which explore the problem of bias in annotation instructions [121] and disconnects in the mental models of annotators and evaluators [25]. Continuing this frontier of work is an opportunity for deep collaboration between the Human-Computer Interaction and Machine Learning communities.

5.2.2 Closing the Loop from Interpretation and Evaluation

While saliency and interpretability toolkits can be critical for understanding edge cases and examining model behavior, all of these toolkits suffer from the existential question of how to

reckon with their results. What is the best way to "fix" a model with unwanted behavior? Of note, IMACS is not exempt from this question—how should one "correct" the glasses bias in the model in the scenario? Researchers and developers often make changes to datasets with the goal of eliminating unwanted biases. Editing the dataset necessarily requires some degree of model retraining. Other techniques have emerged from the ML research community that support "model editing", which can adjust specific outputs with auxiliary models [107, 108] or through more careful resampling and exclusion of data [167]. There is still a wide field of opportunity in this space, particularly for interactive techniques, and interfaces that support non-experts in better understanding and editing their models. This is a phenomenal opportunity for collaboration as well, where direct manipulation or exploratory techniques from HCI [33] could complement algorithmic advances from the ML research community.

5.2.3 Human-Centered Model Explanations

One particularly exciting avenue for future work is to understand the needs that explanations of model behavior fulfill, and to design algorithms, visualizations, and interactions that meet specific, individual needs. The ML interpretabiliy and Explainable AI research communities have highlighted many dimensions of what make explanations effective and truthful in the general case [98]. An inspiring recent work has sought to link design criteria for model explanations with theories of how people understand behavior [80]. Understanding what makes a good explanation, and how one can adjust explanations to match the mental models and contexts of their consumers will remain an important research question, especially as large language models will become the backbones of new user experiences. A possible approach may be to consider a model explanation as a process, rather than a snapshot. There are potential parallels to sensemaking and information foraging [125] in the ways practitioners can begin to understand the mechanisms that influence ML model predictions. For this speculative interaction, what are the best signals that provide clues, and how would one extract them? One important step in this process would be to conduct need finding through contextual inquiries, user evaluations, or participatory design workshops to better understand the disconnects in mental models among different target user groups.

Another parallel track for future work in designing explanations is to understand how the *societal expectations* of model capabilities and failures will evolve. Many consumers of ML applications have learned to expect specific types of failures, such as imperfect machine translations or unexpected social media recommendations. In an optimistic speculative future, consumers of these systems could be given increased agency through more transparent control that can leverage exploration or explanations.

5.2.4 Augmenting Traditional Software Development

One of the most exciting recent trends is the application of large language models to code. Codex [23] and Copilot [45] have already made a significant impact in the software engineering community with their ability to generate code from natural language prompts. Even

general-purpose conversational models have been able to produce and refine code in dialogue [11]. Newer models can perform more advanced tasks, such as bug localization, bug fixing, and generating descriptions of code [163]. These models will enable new programming paradigms (e.g., turning software programming into more of a prompt programming task) which are worth exploring. One opportunity for future work could be to consider interactions with these models as a basis for learning more about API usage or adhering to good design patterns and practices.

5.3 Summary

This thesis introduces works that address critical usability challenges at different stages of the ML application development process. Three systems were presented that help ML application developers conduct searches for ML projects as starting points for their work; help them debug, understand, and fix bugs in their model training code; and better compare and understand the behavioral patterns of their image models. All of these systems instantiate the goal of this thesis, which is to empower ML application developers to identify or create useful and informative structures at each of these stages themselves, by using **exploration** and **explanation**. Ultimately, the goal of this work continues the thread of HCI works that aim to lower the barrier to entry to developing ML applications, and providing tools to help people gain a greater understanding of their applications. A key reflection of this work is that combining design-based approaches, which seek to meet the specific needs of select target users, with general, algorithmic approaches that can provide strong theoretical guarantees, is not only compatible, but necessary. I imagine a future where people with vastly different levels of experience can create and use ML applications as intuitive and expressive media that enhance and spark creativity. The surest way to arrive there is to facilitate deep, meaningful collaborations between the Human-Computer Interaction and Machine Learning research communities, where design-based and axiomatic approaches can be combined to unlock powerful applications that can help elevate and unleash our creative potential.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.

[2] Radhakrishna Achanta et al. "SLIC superpixels". In: *Technical report, EPFL* (June 2010).

[3] Uri Alon et al. "code2vec: Learning Distributed Representations of Code". In: *arXiv:1803.09473 [cs, stat]* (Oct. 2018). arXiv: 1803.09473. URL: http://arxiv.org/abs/1803.09473 (visited on 02/18/2021).

[4] Saleema Amershi et al. "ModelTracker: Redesigning Performance Analysis Tools for Machine Learning". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: Association for Computing Machinery, 2015, pp. 337–346. ISBN: 9781450331456. DOI: 10.1145/2702123.2702509. URL: https://doi.org/10.1145/2702123.2702509.

[5] Saleema Amershi et al. "ModelTracker: Redesigning Performance Analysis Tools for Machine Learning". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: Association for Computing Machinery, 2015, pp. 337–346. ISBN: 9781450331456. DOI: 10.1145/2702123.2702509. URL: https://doi.org/10.1145/2702123.2702509.

[6] Saleema Amershi et al. "Software Engineering for Machine Learning: A Case Study". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042. URL: https://doi.org/10.1109/ICSE-SEIP.2019.00042.

[7] Kanav Anand et al. "Black Magic in Deep Learning: How Human Skill Impacts Network Training". In: *The British Machine Vision Conference* (2020).

[8] Marc Andreessen. "Why Software Is Eating The World". en-US. In: *Wall Street Journal* (Aug. 2011). ISSN: 0099-9660. URL: https://online.wsj.com/article/SB10001424053111903480904576512250915629460.html (visited on 10/11/2022).

[9] Keith Andrews et al. “The InfoSky Visual Explorer: Exploiting Hierarchical Structure and Document Similarities”. In: *Information Visualization* 1.3/4 (Dec. 2002), pp. 166–181. ISSN: 1473-8716. DOI: 10.1057/palgrave.ivs.9500023. URL: https://doi.org/10.1057/palgrave.ivs.9500023.

[10] *Anyscale - Effortlessly develop, scale and deploy AI, at any scale — Anyscale*. URL: https://www.anyscale.com/ (visited on 10/12/2022).

[11] Jacob Austin et al. *Program Synthesis with Large Language Models*. 2021. DOI: 10.48550/ARXIV.2108.07732. URL: https://arxiv.org/abs/2108.07732.

[12] J. Bergstra et al. “Algorithms for Hyper-Parameter Optimization”. In: *NIPS*. 2011.

[13] Tolga Bolukbasi et al. *An Interpretability Illusion for BERT*. 2021. DOI: 10.48550/ARXIV.2104.07143. URL: https://arxiv.org/abs/2104.07143.

[14] Houssem Ben Braiek and Foutse Khomh. *TFCheck : A TensorFlow Library for Detecting Training Issues in Neural Network Programs*. 2019. arXiv: 1909.02562 [cs.LG].

[15] Joel Brandt et al. “Example-Centric Programming: Integrating Web Search into the Development Environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, pp. 513–522. ISBN: 9781605589299. DOI: 10.1145/1753326.1753402. URL: https://doi.org/10.1145/1753326.1753402.

[16] Joel Brandt et al. “Example-Centric Programming: Integrating Web Search into the Development Environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, pp. 513–522. ISBN: 9781605589299. DOI: 10.1145/1753326.1753402. URL: https://doi.org/10.1145/1753326.1753402.

[17] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[18] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[19] Steve Burbeck. “Applications programming in smalltalk-80: how to use model-view-controller (mvc)”. In: 1987.

[20] Carrie J Cai et al. “Onboarding Materials as Cross-Functional Boundary Objects for Developing AI Assistants”. In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI EA ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380959. DOI: 10.1145/3411763.3443435. URL: https://doi.org/10.1145/3411763.3443435.

[21] Carrie J. Cai and Philip J. Guo. "Software Developers Learning Machine Learning: Motivations, Hurdles, and Desires". In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Oct. 2019, pp. 25–34. DOI: 10.1109/VLHCC.2019.8818751.

[22] Shanqing Cai. *Debug TensorFlow Models with tfdbg*. Feb. 2017. URL: https://developers.googleblog.com/2017/02/debug-tensorflow-models-with-tfdbg.html.

[23] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. DOI: 10.48550/ARXIV.2107.03374. URL: https://arxiv.org/abs/2107.03374.

[24] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.

[25] Elizabeth Clark et al. *All That's 'Human' Is Not Gold: Evaluating Human Evaluation of Generated Text*. 2021. DOI: 10.48550/ARXIV.2107.00061. URL: https://arxiv.org/abs/2107.00061.

[26] Andy Coenen and Adam Pearce. "Understanding UMAP". In: *URL https://pair-code.github.io/understanding-umap* (2019).

[27] Alexis Conneau et al. "Very Deep Convolutional Networks for Text Classification". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*. Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pp. 1107–1116. URL: https://aclanthology.org/E17-1104 (visited on 09/16/2022).

[28] D. Cubranic et al. "Hipikat: a project memory for software development". In: *IEEE Transactions on Software Engineering* 31.6 (June 2005). Conference Name: IEEE Transactions on Software Engineering, pp. 446–465. ISSN: 1939-3520. DOI: 10.1109/TSE.2005.71.

[29] Douglass R. Cutting et al. "Scatter/Gather: A Cluster-Based Approach to Browsing Large Document Collections". In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '92. Copenhagen, Denmark: Association for Computing Machinery, 1992, pp. 318–329. ISBN: 0897915232. DOI: 10.1145/133160.133214. URL: https://doi.org/10.1145/133160.133214.

[30] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[31] Emily Denton et al. "Detecting Bias with Generative Counterfactual Face Attribute Augmentation". In: *CoRR* abs/1906.06439 (2019). arXiv: 1906.06439. URL: http://arxiv.org/abs/1906.06439.

[32] Daniel Drew et al. "The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: Association for Computing Machinery, 2016, pp. 677–686. ISBN: 9781450341899. DOI: 10.1145/2984511.2984566. URL: https://doi.org/10.1145/2984511.2984566.

[33] Jerry Alan Fails and Dan R. Olsen. "Interactive Machine Learning". In: *Proceedings of the 8th International Conference on Intelligent User Interfaces*. IUI '03. Miami, Florida, USA: Association for Computing Machinery, 2003, pp. 39–45. ISBN: 1581135866. DOI: 10.1145/604045.604056. URL: https://doi.org/10.1145/604045.604056.

[34] William Falcon et al. *PyTorchLightning/pytorch-lightning: 0.7.6 release*. Version 0.7.6. May 2020. DOI: 10.5281/zenodo.3828935. URL: https://doi.org/10.5281/zenodo.3828935.

[35] Cristian Felix, Aritra Dasgupta, and Enrico Bertini. "The Exploratory Labeling Assistant: Mixed-Initiative Label Curation with Large Document Collections". In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 153–164. ISBN: 9781450359481. DOI: 10.1145/3242587.3242596. URL: https://doi.org/10.1145/3242587.3242596.

[36] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *arXiv:2002.08155 [cs]* (Sept. 2020). arXiv: 2002.08155. URL: http://arxiv.org/abs/2002.08155 (visited on 03/09/2021).

[37] Rebecca Fiebrink and Perry R Cook. "The Wekinator: a system for real-time, interactive machine learning in music". In: *Proceedings of The Eleventh International Society for Music Information Retrieval Conference (ISMIR 2010)(Utrecht)*. 2010.

[38] Ruth C Fong and Andrea Vedaldi. "Interpretable explanations of black boxes by meaningful perturbation". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 3429–3437.

[39] Adam Fourney and Meredith Ringel Morris. "Enhancing Technical Q&A Forums with CiteHistory". In: *Proceedings of ICWSM 2013*. You can download the CiteHistory plugin at http://research.microsoft.com/en-us/um/redmond/projects/citehistory/. AAAI, July 2013. URL: https://www.microsoft.com/en-us/research/publication/enhancing-technical-qa-forums-with-citehistory/.

[40] Rolando Garcia et al. *flor*. 2019. URL: https://github.com/ucbrise/flor.

[41] Timnit Gebru et al. "Datasheets for Datasets". In: *Commun. ACM* 64.12 (Nov. 2021), pp. 86–92. ISSN: 0001-0782. DOI: 10.1145/3458723. URL: https://doi.org/10.1145/3458723.

[42] Mehrdad Ghadiri, Samira Samadi, and Santosh Vempala. "Socially Fair K-Means Clustering". In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 438–448. ISBN: 9781450383097. DOI: 10.1145/3442188.3445906. URL: https://doi.org/10.1145/3442188.3445906.

[43] Amirata Ghorbani et al. "Towards Automatic Concept-based Explanations". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/77d2afcb31f6493e350fca61764efb9a-Paper.pdf.

[44] Leilani H. Gilpin et al. *Explaining Explanations: An Overview of Interpretability of Machine Learning*. 2018. arXiv: 1806.00069 [cs.AI].

[45] *GitHub Copilot · Your AI pair programmer*. en. URL: https://github.com/features/copilot (visited on 10/14/2022).

[46] Elena L. Glassman et al. "Visualizing API Usage Examples at Scale". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–12. ISBN: 9781450356206. DOI: 10.1145/3173574.3174154. URL: https://doi.org/10.1145/3173574.3174154.

[47] Michael Gleicher et al. *Boxer: Interactive Comparison of Classifier Results*. 2020. arXiv: 2004.07964 [cs.HC].

[48] Max Goldman and Robert C. Miller. "Codetrail: Connecting source code and web resources". In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 2008, pp. 65–72. DOI: 10.1109/VLHCC.2008.4639060.

[49] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[50] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[51] Tovi Grossman, George Fitzmaurice, and Ramtin Attar. "A survey of software learnability: metrics, methodologies and guidelines". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2009, pp. 649–658.

[52] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep Code Search". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 933–944. DOI: 10.1145/3180155.3180167.

[53] Björn Hartmann et al. "What Would Other Programmers Do: Suggesting Solutions to Error Messages". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, pp. 1019–1028. ISBN: 9781605589299. DOI: 10.1145/1753326.1753478. URL: https://doi.org/10.1145/1753326.1753478.

[54] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[55] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[56] Kaiming He et al. *Identity Mappings in Deep Residual Networks*. 2016. arXiv: 1603.05027 [cs.CV].

[57] Andrew Head. “Social health cues developers use when choosing open source packages”. en. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle WA USA: ACM, Nov. 2016, pp. 1133–1135. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983973. URL: https://dl.acm.org/doi/10.1145/2950290.2983973 (visited on 09/15/2022).

[58] Andrew Head et al. “Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376798. URL: https://doi.org/10.1145/3313831.3376798.

[59] Andrew Head et al. “Managing Messes in Computational Notebooks”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, pp. 1–12. ISBN: 9781450359702. DOI: 10.1145/3290605.3300500. URL: https://doi.org/10.1145/3290605.3300500.

[60] Andrew Head et al. “Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Oct. 2015, pp. 3–12. DOI: 10.1109/VLHCC.2015.7356972.

[61] Marti A. Hearst. *Search User Interfaces*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521113792.

[62] Patrick Helber et al. “EuroSAT: A Novel Dataset and Deep Learning Benchmark for Land Use and Land Cover Classification”. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.7 (2019), pp. 2217–2226. DOI: 10.1109/JSTARS.2019.2918242.

[63] Shawn Hershey et al. “CNN architectures for large-scale audio classification”. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2017), pp. 131–135.

[64] Charles Hill et al. “Trials and tribulations of developers of intelligent systems: A field study”. In: *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Sept. 2016, pp. 162–170. DOI: 10.1109/VLHCC.2016.7739680.

[65] Andreas Hinterreiter et al. "ConfusionFlow: A model-agnostic visualization for temporal analysis of classifier confusion". In: *IEEE Transactions on Visualization and Computer Graphics* (2020), pp. 1–1. ISSN: 2160-9306. DOI: 10.1109/tvcg.2020.3012063. URL: http://dx.doi.org/10.1109/TVCG.2020.3012063.

[66] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. "Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers". In: *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. UIST '07. Newport, Rhode Island, USA: Association for Computing Machinery, 2007, pp. 13–22. ISBN: 9781595936790. DOI: 10.1145/1294211.1294216. URL: https://doi.org/10.1145/1294211.1294216.

[67] Fred Hohman et al. "Gamut: A Design Probe to Understand How Data Scientists Understand Machine Learning Models". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019. ISBN: 9781450359702. DOI: 10.1145/3290605.3300809. URL: https://doi.org/10.1145/3290605.3300809.

[68] Fred Hohman et al. "Understanding and Visualizing Data Iteration in Machine Learning". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376177. URL: https://doi.org/10.1145/3313831.3376177.

[69] Fred Hohman et al. "Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers". In: *IEEE Transactions on Visualization and Computer Graphics* 25.8 (2019), pp. 2674–2693. DOI: 10.1109/TVCG.2018.2843369.

[70] Jeremy Howard and Sylvain Gugger. "Fastai: A Layered API for Deep Learning". In: *Information* 11.2 (Feb. 2020), p. 108. ISSN: 2078-2489. DOI: 10.3390/info11020108. URL: http://dx.doi.org/10.3390/info11020108.

[71] Forrest Huang et al. "Creating User Interface Mock-ups from High-Level Text Descriptions with Deep-Learning Models". In: *CoRR* abs/2110.07775 (2021). arXiv: 2110.07775. URL: https://arxiv.org/abs/2110.07775.

[72] Forrest Huang et al. "Scones: Towards Conversational Authoring of Sketches". In: *Proceedings of the 25th International Conference on Intelligent User Interfaces*. IUI '20. Cagliari, Italy: Association for Computing Machinery, 2020, pp. 313–323. ISBN: 9781450371186. DOI: 10.1145/3377325.3377485. URL: https://doi.org/10.1145/3377325.3377485.

[73] Nargiz Humbatova et al. *Taxonomy of Real Faults in Deep Learning Systems*. 2019. arXiv: 1910.11015 [cs.SE].

[74] Hamel Husain et al. "CodeSearchNet challenge: Evaluating the state of semantic code search". In: *arXiv preprint arXiv:1909.09436* (2019).

[75] Ben Hutchinson et al. "Towards Accountability for Machine Learning Datasets: Practices from Software Engineering and Infrastructure". In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 560–575. ISBN: 9781450383097. DOI: 10.1145/3442188.3445918. URL: https://doi.org/10.1145/3442188.3445918.

[76] Apple Inc. *Apple Create ML*. 2019. URL: https://developer.apple.com/machine-learning/create-ml/.

[77] Databricks Inc. *MLFlow*. 2019. URL: https://mlflow.org/.

[78] Google Inc. *Know Your Data*. en. May 2021. URL: https://knowyourdata.withgoogle.com/ (visited on 09/30/2022).

[79] Md Johirul Islam et al. "A Comprehensive Study on Deep Learning Bug Characteristics". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 510–520. ISBN: 9781450355728. DOI: 10.1145/3338906.3338955. URL: https://doi.org/10.1145/3338906.3338955.

[80] Alon Jacovi et al. *Diagnosing AI Explanation Methods with Folk Concepts of Behavior*. 2022. DOI: 10.48550/ARXIV.2201.11239. URL: https://arxiv.org/abs/2201.11239.

[81] Andrew Janowczyk and Anant Madabhushi. "Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases". In: *Journal of pathology informatics* 7 (2016).

[82] S. C. Johnson. "Lint, a C Program Checker". In: *Technical Report*. Bell Telephone Laboratories, 1978, pp. 78–1273.

[83] Minsuk Kahng, Dezhi Fang, and Duen Horng (Polo) Chau. "Visual Exploration of Machine Learning Results Using Data Cube Analysis". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450342070. DOI: 10.1145/2939502.2939503. URL: https://doi.org/10.1145/2939502.2939503.

[84] Aditya Kanade et al. "Learning and Evaluating Contextual Embedding of Source Code". In: *arXiv:2001.00059 [cs]* (Aug. 2020). arXiv: 2001.00059. URL: http://arxiv.org/abs/2001.00059 (visited on 03/09/2021).

[85] Andrei Kapishnikov et al. "Guided Integrated Gradients: An Adaptive Path Method for Removing Noise". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2021, pp. 5050–5058.

[86] Andrei Kapishnikov et al. *XRAI: Better Attributions Through Regions*. 2019. arXiv: 1906.02825 [cs.CV].

[87] Andrej Karpathy. *A Recipe for Training Neural Networks*. Apr. 2019. URL: https://karpathy.github.io/2019/04/25/recipe/.

[88] Andrej Karpathy. *Software 2.0*. en. Nov. 2017. URL: https://karpathy.medium.com/software-2-0-a64152b37c35 (visited on 10/11/2022).

[89] Andrej Karpathy. "Training Neural Networks, Part 1". In: *Convolutional Neural Networks for Visual Recognition. Lecture Slides* (Jan. 2016). URL: http://cs231n.stanford.edu/2016/syllabus.html.

[90] Jun Kato, Sean McDirmid, and Xiang Cao. "DejaVu: integrated support for developing interactive camera-based programs". In: *Proceedings of the 25th annual ACM symposium on User interface software and technology*. 2012, pp. 189–196.

[91] Been Kim et al. *Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)*. 2017. arXiv: 1711.11279 [stat.ML].

[92] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[93] Amy J. Ko and Brad A. Myers. "Finding Causes of Program Output with the Java Whyline". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 1569–1578. ISBN: 9781605582467. DOI: 10.1145/1518701.1518942. URL: https://doi.org/10.1145/1518701.1518942.

[94] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[95] Alina Kuznetsova et al. "The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale". In: *IJCV* (2020).

[96] Philippe Laban et al. "The Summary Loop: Learning to Write Abstractive Summaries Without Examples". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 5135–5150. DOI: 10.18653/v1/2020.acl-main.460. URL: https://aclanthology.org/2020.acl-main.460.

[97] Lezhi Li and Yang Wang. *Manifold: A Model-Agnostic Visual Debugging Tool for Machine Learning at Uber*. Aug. 2019. URL: https://eng.uber.com/manifold/.

[98] Zachary Chase Lipton. "The Myth of Model Interpretability". In: *CoRR* abs / 1606.03490 (2016). arXiv: 1606.03490. URL: http://arxiv.org/abs/1606.03490.

[99] Google LLC. *Machine Learning Crash Course with TensorFlow APIs*. 2020. URL: https://developers.google.com/machine-learning/crash-course.

[100] Shuai Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *CoRR* abs/2102.04664 (2021).

[101] Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. "Eyepatch: prototyping camera-based interaction through examples". In: *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 2007, pp. 33–42.

[102] Will McGrath et al. "Bifröst: Visualizing and Checking Behavior of Embedded Systems across Hardware and Software". In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. Québec City, QC, Canada: Association for Computing Machinery, 2017, pp. 299–310. ISBN: 9781450349819. DOI: 10.1145/3126594.3126658. URL: https://doi.org/10.1145/3126594.3126658.

[103] David A. Mellis et al. "Machine Learning for Makers: Interactive Sensor Data Classification Based on Augmented Code Examples". In: *Proceedings of the 2017 Conference on Designing Interactive Systems*. DIS '17. Edinburgh, United Kingdom: Association for Computing Machinery, 2017, pp. 1213–1225. ISBN: 9781450349222. DOI: 10.1145/3064663.3064735. URL: https://doi.org/10.1145/3064663.3064735.

[104] Microsoft. *Automate code completions tailored to your codebase with IntelliCode Team completions*. 2020. URL: https://github.com/microsoft/vs-intellicode.

[105] *Midjourney*. en. URL: https://www.midjourney.com/home/ (visited on 10/11/2022).

[106] Riccardo Miotto et al. "Deep learning for healthcare: review, opportunities and challenges". In: *Briefings in Bioinformatics* 19.6 (May 2017), pp. 1236–1246. ISSN: 1477-4054. DOI: 10.1093/bib/bbx044. eprint: https://academic.oup.com/bib/article-pdf/19/6/1236/27119191/bbx044.pdf. URL: https://doi.org/10.1093/bib/bbx044.

[107] Eric Mitchell et al. *Fast Model Editing at Scale*. 2021. DOI: 10.48550/ARXIV.2110.11309. URL: https://arxiv.org/abs/2110.11309.

[108] Eric Mitchell et al. *Memory-Based Model Editing at Scale*. 2022. DOI: 10.48550/ARXIV.2206.06520. URL: https://arxiv.org/abs/2206.06520.

[109] Margaret Mitchell et al. "Model Cards for Model Reporting". In: *Proceedings of the Conference on Fairness, Accountability, and Transparency*. FAT* '19. Atlanta, GA, USA: Association for Computing Machinery, 2019, pp. 220–229. ISBN: 9781450361255. DOI: 10.1145/3287560.3287596. URL: https://doi.org/10.1145/3287560.3287596.

[110] Philipp Moritz et al. *Ray: A Distributed Framework for Emerging AI Applications*. 2017. DOI: 10.48550/ARXIV.1712.05889. URL: https://arxiv.org/abs/1712.05889.

[111] Sugeerth Murugesan et al. "DeepCompare: Visual and Interactive Comparison of Deep Learning Model Performance". In: *IEEE Comput. Graph. Appl.* 39.5 (Sept. 2019), pp. 47–59. ISSN: 0272-1716. DOI: 10.1109/MCG.2019.2919033. URL: https://doi.org/10.1109/MCG.2019.2919033.

[112] Sugeerth Murugesan et al. "DeepCompare: Visual and Interactive Comparison of Deep Learning Model Performance". In: *IEEE Computer Graphics and Applications* PP (May 2019), pp. 1–1. DOI: 10.1109/MCG.2019.2919033.

[113] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[114] Shweta Narkar et al. "Model LineUpper: Supporting Interactive Model Comparison at Multiple Levels for AutoML". In: *26th International Conference on Intelligent User Interfaces*. IUI '21. College Station, TX, USA: Association for Computing Machinery, 2021, pp. 170–174. ISBN: 9781450380171. DOI: 10.1145/3397481.3450658. URL: https://doi.org/10.1145/3397481.3450658.

[115] Soroush Nasiriany et al. *A Comprehensive Guide to Machine Learning*. Nov. 2019.

[116] *NVIDIA DLSS 2.0: A Big Leap In AI Rendering*. URL: https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/.

[117] Augustus Odena et al. "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, June 2019, pp. 4901–4911. URL: http://proceedings.mlr.press/v97/odena19a.html.

[118] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. "Feature Visualization". In: *Distill* (2017). https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

[119] Savannah Ostrowski. *Announcing Pylance: Fast, feature-rich language support for Python in Visual Studio Code*. 2020. URL: https://devblogs.microsoft.com/python/announcing-pylance-fast-feature-rich-language-support-for-python-in-visual-studio-code/.

[120] Google PAIR. *FACETS*. 2017. URL: https://pair-code.github.io/facets/.

[121] Mihir Parmar et al. *Don't Blame the Annotator: Bias Already Starts in the Annotation Instructions*. 2022. DOI: 10.48550/ARXIV.2205.00415. URL: https://arxiv.org/abs/2205.00415.

[122] Kayur Patel et al. "Gestalt: Integrated Support for Implementation and Analysis in Machine Learning". In: *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology*. UIST '10. New York, New York, USA: Association for Computing Machinery, 2010, pp. 37–46. ISBN: 9781450302715. DOI: 10.1145/1866029.1866038. URL: https://doi.org/10.1145/1866029.1866038.

[123] Kayur Patel et al. "Investigating 1Statistical Machine Learning as a Tool for Software Development". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '08. Florence, Italy: Association for Computing Machinery, 2008, pp. 667–676. ISBN: 9781605580111. DOI: 10.1145/1357054.1357160. URL: https://doi.org/10.1145/1357054.1357160.

[124] Neel Patel, Martin Strobel, and Yair Zick. "High Dimensional Model Explanations: An Axiomatic Approach". In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 401–411. ISBN: 9781450383097. DOI: 10.1145/3442188.3445903. URL: https://doi.org/10.1145/3442188.3445903.

[125] Peter Pirolli and Stuart Card. "Information foraging". In: *Psychological Review* 106 (1999). Place: US Publisher: American Psychological Association, pp. 643–675. ISSN: 1939-1471. DOI: 10.1037/0033-295X.106.4.643.

[126] Peter Pirolli and Stuart Card. "The Sensemaking Process and Leverage Points for Analyst Technology as Identified Through Cognitive Task Analysis". In: *Proceedings of the International Conference on Intelligence Analysis*. Vol. 5. May 2005.

[127] Peter Pirolli et al. "Scatter/Gather Browsing Communicates the Topic Structure of a Very Large Text Collection". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '96. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1996, pp. 213–220. ISBN: 0897917774. DOI: 10.1145/238386.238489. URL: https://doi.org/10.1145/238386.238489.

[128] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. DOI: 10.48550/ARXIV.2204.06125. URL: https://arxiv.org/abs/2204.06125.

[129] Benjamin Recht et al. "Do ImageNet Classifiers Generalize to ImageNet?" In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, June 2019, pp. 5389–5400. URL: https://proceedings.mlr.press/v97/recht19a.html.

[130] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778. URL: https://doi.org/10.1145/2939672.2939778.

[131] Adam Roberts et al. "Scaling Up Models and Data with `t5x` and `seqio`". In: *arXiv preprint arXiv:2203.17189* (2022). URL: https://arxiv.org/abs/2203.17189.

[132] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2021. DOI: 10.48550/ARXIV.2112.10752. URL: https://arxiv.org/abs/2112.10752.

[133] Xin Rong et al. "CodeMend: Assisting Interactive Programming with Bimodal Embedding". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: Association for Computing Machinery, 2016, pp. 247–258. ISBN: 9781450341899. DOI: 10.1145/2984511.2984544. URL: https://doi.org/10.1145/2984511.2984544.

[134] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". en. In: *arXiv:1409.0575 [cs]* (Sept. 2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575 (visited on 03/18/2019).

[135] Daniel M. Russell et al. "The Cost Structure of Sensemaking". In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: Association for Computing Machinery, 1993, pp. 269–276. ISBN: 0897915755. DOI: 10.1145/169059.169209. URL: https://doi.org/10.1145/169059.169209.

[136] Saksham Sachdev et al. "Retrieval on source code: a neural code search". en. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. Philadelphia PA USA: ACM, June 2018, pp. 31–41. ISBN: 978-1-4503-5834-7. DOI: 10.1145/3211346.3211353. URL: https://dl.acm.org/doi/10.1145/3211346.3211353 (visited on 07/17/2021).

[137] Chitwan Saharia et al. *Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding*. 2022. DOI: 10.48550/ARXIV.2205.11487. URL: https://arxiv.org/abs/2205.11487.

[138] Eldon Schoop, Forrest Huang, and Bjoern Hartmann. "UMLAUT: Debugging Deep Learning Programs Using Program Structure and Model Behavior". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445538. URL: https://doi.org/10.1145/3411764.3445538.

[139] Eldon Schoop, Forrest Huang, and Björn Hartmann. "SCRAM: Simple Checks for Realtime Analysis of Model Training for Non-Expert ML Programmers". In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–10. ISBN: 9781450368193. DOI: 10.1145/3334480.3382879. URL: https://doi.org/10.1145/3334480.3382879.

[140] Eldon Schoop et al. "IMACS: Image Model Attribution Comparison Summaries". In: *CoRR* abs/2201.11196 (2022). arXiv: 2201.11196. URL: https://arxiv.org/abs/2201.11196.

[141] Ramprasaath R Selvaraju et al. "Grad-cam: Visual explanations from deep networks via gradient-based localization". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 618–626.

[142] Shital Shah, Roland Fernandez, and Steven Drucker. "A System for Real-Time Interactive Analysis of Deep Learning Training". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS '19. Valencia, Spain: Association for Computing Machinery, 2019. ISBN: 9781450367455. DOI: 10.1145/3319499.3328231. URL: https://doi.org/10.1145/3319499.3328231.

[143] Lloyd S Shapley. "A value for n-person games". In: *Contributions to the Theory of Games* 2.28 (1953), pp. 307–317.

[144] Jonathan R. Shewchuk. *Concise Machine Learning*. May 2020. URL: https://people.eecs.berkeley.edu/~jrs/papers/machlearn.pdf.

[145] Ben Shneiderman. "Creativity Support Tools: Accelerating Discovery and Innovation". In: *Commun. ACM* 50.12 (Dec. 2007), pp. 20–32. ISSN: 0001-0782. DOI: 10.1145/1323688.1323689. URL: https://doi.org/10.1145/1323688.1323689.

[146] Daniel Smilkov et al. *Embedding Projector: Interactive Visualization and Interpretation of Embeddings*. 2016. DOI: 10.48550/ARXIV.1611.05469. URL: https://arxiv.org/abs/1611.05469.

[147] Daniel Smilkov et al. *TensorFlow.js: Machine Learning for the Web and Beyond*. 2019. DOI: 10.48550/ARXIV.1901.05350. URL: https://arxiv.org/abs/1901.05350.

[148] Kacper Sokol and Peter Flach. "Explainability Fact Sheets: A Framework for Systematic Assessment of Explainable Approaches". In: *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. FAT* '20. Barcelona, Spain: Association for Computing Machinery, 2020, pp. 56–67. ISBN: 9781450369367. DOI: 10.1145/3351095.3372870. URL: https://doi.org/10.1145/3351095.3372870.

[149] John T Stasko, Marc H Brown, and Blaine A Price. *Software Visualization*. MIT press, 1997.

[150] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. URL: https://www.aclweb.org/anthology/P19-1355.

[151] Dong Sun et al. "DFSeer: A Visual Analytics Approach to Facilitate Model Selection for Demand Forecasting". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376866. URL: https://doi.org/10.1145/3313831.3376866.

[152] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 3319–3328.

[153] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: June 2016. DOI: 10.1109/CVPR.2016.308.

[154] The TensorFlow Team. *Flowers*. Jan. 2019. URL: http://download.tensorflow.org/example_images/flower_photos.tgz.

[155] Rachel Thomas and David Uminsky. *The Problem with Metrics is a Fundamental Problem for AI*. 2020. arXiv: 2002.08512 [cs.CY].

[156] Josh Tobin. *Troubleshooting Deep Neural Networks: A Field Guide to Fixing Your Model*. 2019. URL: http://josh-tobin.com/troubleshooting-deep-neural-networks.html.

[157] Jason Tsay et al. "AIMMX: Artificial Intelligence Model Metadata Extractor". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 81–92. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387448. URL: https://doi.org/10.1145/3379597.3387448 (visited on 07/16/2021).

[158] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[159] Manasi Vartak et al. "M¡span class="smallcaps SmallerCapital"¿odel¡/span¿DB: A System for Machine Learning Model Management". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450342070. DOI: 10.1145/2939502.2939516. URL: https://doi.org/10.1145/2939502.2939516.

[160] Matthew Veres and Medhat Moussa. "Deep learning for intelligent transportation systems: A survey of emerging trends". In: *IEEE Transactions on Intelligent transportation systems* (2019).

[161] Alex Wang et al. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355. DOI: 10.18653/v1/W18-5446. URL: https://aclanthology.org/W18-5446.

[162] Angelina Wang, Arvind Narayanan, and Olga Russakovsky. "REVISE: A Tool for Measuring and Mitigating Bias in Visual Datasets". In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Cham: Springer International Publishing, 2020, pp. 733–751. ISBN: 978-3-030-58580-8.

[163] Yue Wang et al. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685. URL: https://aclanthology.org/2021.emnlp-main.685.

[164] Robert Stuart Weiss. *Learning from strangers: the art and method of qualitative interview studies*. Free Press, 1995.

[165] James Wexler et al. “The What-If Tool: Interactive Probing of Machine Learning Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 56–65. DOI: 10.1109/TVCG.2019.2934619.

[166] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2019. DOI: 10.48550/ARXIV.1910.03771. URL: https://arxiv.org/abs/1910.03771.

[167] Ga Wu, Masoud Hashemi, and Christopher Srinivasa. *PUMA: Performance Unchanged Model Augmentation for Training Data Removal*. 2022. DOI: 10.48550/ARXIV.2203.00846. URL: https://arxiv.org/abs/2203.00846.

[168] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[169] Litao Yan, Elena L. Glassman, and Tianyi Zhang. “Visualizing Examples of Deep Neural Networks at Scale”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445654. URL: https://doi.org/10.1145/3411764.3445654.

[170] Litao Yan et al. “Concept-Annotated Examples for Library Comparison”. In: *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. UIST '22. Bend, Oregon, USA: Association for Computing Machinery, 2022. ISBN: 9781450393201. DOI: 10.1145/3526113.3545647. URL: https://doi.org/10.1145/3526113.3545647.

[171] Yi Yang and Shawn Newsam. “Bag-of-Visual-Words and Spatial Extensions for Land-Use Classification”. In: *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '10. San Jose, California: Association for Computing Machinery, 2010, pp. 270–279. ISBN: 9781450304283. DOI: 10.1145/1869790.1869829. URL: https://doi.org/10.1145/1869790.1869829.

[172] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. “Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training”. In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 126–139. ISBN: 9781450375146. DOI: 10.1145/3379337.3415890. URL: https://doi.org/10.1145/3379337.3415890.

[173] Jiahui Yu et al. *Scaling Autoregressive Models for Content-Rich Text-to-Image Generation*. 2022. DOI: 10.48550/ARXIV.2206.10789. URL: https://arxiv.org/abs/2206.10789.

[174] Daniel Zhang et al. *The AI Index 2021 Annual Report*. Tech. rep. Stanford, CA: AI Index Steering Committee, Human-Centered AI Institute, Stanford University, Mar. 2021.

[175] Richard Zhang et al. "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric". In: *CVPR*. 2018.

[176] Tianyi Zhang et al. "An Empirical Study of Common Challenges in Developing Deep Learning Applications". In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 2019, pp. 104–115. DOI: 10.1109/ISSRE.2019.00020.

[177] Xiaoyi Zhang et al. "Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445186. URL: https://doi.org/10.1145/3411764.3445186.

[178] Yuhao Zhang et al. "An Empirical Study on TensorFlow Program Bugs". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 129–140. ISBN: 9781450356992. DOI: 10.1145/3213846.3213866. URL: https://doi.org/10.1145/3213846.3213866.