Lowering the Barrier to Applying Machine Learning

Kayur Dushyant Patel

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Program Authorized to Offer Degree:

Computer Science and Engineering

University of Washington

Abstract

Lowering the Barrier to Applying Machine Learning

Kayur Dushyant Patel

Co-Chairs of the Supervisory Committee:

Associate Professor James A. Fogarty
Computer Science and Engineering

Professor James A. Landay
Computer Science and Engineering

Data is driving the future of computation: analysis, visualization, and learning algorithms power systems that help us diagnose cancer, live sustainably, and understand the universe. Yet, the data explosion has outstripped our tools to process it, leaving a gap between powerful new algorithms and what real programmers can apply in practice.

I examine how data affects the way we program. Specifically, this dissertation focuses on using machine learning algorithms to train a model. I found that the key barrier to adoption is not a poor understanding of the machine learning algorithms themselves, but rather a poor understanding of the process for applying those algorithms and insufficient tool support for that process. I have created new programming and analysis tools that support programmers by helping them (1) implement machine learning systems and analyze results, (2) debug data, and (3) design and track experiments.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ANALYSIS: the act of understanding the data, the code, and the relationships between the data and the code.

AUTHOR: to program the behavior of a function by writing code.

CLASSIFICATION PIPELINE: the data flow that is constructed for turning raw data into a model using a classification algorithm.

FUNCTION: a mapping from inputs to outputs.

IMPLEMENTATION: the act of writing code to define the steps in the classification pipeline and providing data to train a model.

MACHINE LEARNING PROCESS: the process of training a model using machine learning – involves the tasks of implementation and analysis.

MODEL: the output of a machine learning algorithm. A model is a function learned from the data.

TRAIN: to program the behavior of a function (i.e., model) by writing code and providing data.

ACKNOWLEDGMENTS

It is impossible to write this section. There are intangibles at play here, and any attempt to enumerate and describe the people who have contributed to my success (loosely defined) will be incomplete. Regardless, an attempt must be made. Here's mine.

Research doesn't happen in a vacuum. (That is unless you're an experimental astrophysicist.) The people that you surround yourself with play a huge role in both your research and your development as a human being. I've been privileged to be surrounded by some amazing people.

First, I've had the dumb luck of being advised by two awesome human beings (or, at least, very convincing robots), both named James Anthony, both with PhDs from CMU, both with a tendency to drink a bit too much at conferences. Under their wings, I have learned how to be passionate about a problem, how to think big, how to decompose big ideas into manageable pieces, and how to hold my liquor.

James Fogarty has a rare gift among academics: the ability to listen. It allows him to absorb half-formed ideas and understand the insights behind them, and it allows him to take criticism from his students and change his behavior. It makes the students who work with him feel like they are working with a true collaborator instead of a boss. Listening requires both patience and humility – you have to check your ego at the door. James's ability to listen is a skill I both admire and envy.

James Landay has taught me to look at the big picture and to take risks. Landay invests in people rather than ideas (at times to his detriment). He took a risk on me, allowed me to explore ideas, and invested heavily when I found a hard problem that I was passionate about. He doesn't let his students sell themselves short and will never pull punches. He protects his students' interests, and he fights tooth and nail for them. This is a profoundly unselfish act, especially in a business that encourages you to push *your* intellectual agenda above all else.

The folks on my dissertation committee have provided sage advice over the years. Steve Drucker proves that you can be both a successful researcher and the nicest person in the world. Andy Ko has forced me to focus my ideas though his socratic advising style and insistence on precision. I've taken advantage of David Notkin's open door, comforting words, and superhuman empathy in many times of angst. Dan Weld has lent me his immense intellect in times when I needed to detangle my thoughts.

During my time at the University of Washington, I have had the pleasure of working with a number of great researchers on a variety of cool projects. I'd like to thank my co-authors Saleema Amershi, Naomi Bancroft, Mike Chen, Steven Drucker, James Fogarty, Susumu Harada, Beverly Harrison, Raphael Hoffmann, Ashish Kapoor, Andy Ko, James Landay, Jon Lester, Scott Saponas, Ian Smith, Desney Tan, Dan Weld, and Fei Wu. I have enjoyed tackling hard problems, sharing ideas, and building solutions with you.

Most folks will tell you that your peers, the people in your research group, can be more instrumental than your advisors at helping you develop intellectually. Being around smart people makes you smarter, and as part of two talented research groups, I've had the privilege of being around very smart people. In the Landay group, I found a cohort of young students eager to build a vibrant HCI community. I'd like to thank Kate Everett for keeping Landay meetings on track; Jon Froehlich for his charisma, aesthetic sense, and willingness to teach a young grad student how to build a GUI; Susumu Harada for providing a shining example of both staggering humility and exceptional research talent; Scott Saponas for his corny humor and big heart; and Michael Toomim for his creativity, passion, and willingness to shirk the system.

The Fogarty research group was a different beast. As an older student, I was able to see young students enter and grow. It was a great experience. I'd like to thank Saleema Amershi for going on adventures with me and not killing me when we decided to work together; Morgan Dixon for being a good (read: non-alcoholic) influence on conference trips, for running 6 miles around Greenlake, and for his unpretentious approach to handling success; Katie Kuksenok for her energy and eccentricity; and Ryder Ziola for his precision and humor.

There are also older students outside of my research groups who have been instrumental in guiding me through the years. I would like to thank Eytan Adar, Jeff Bigham, Krzysztof Gajos, and Adrien Treuille for being great role models and for providing great advice.

No UW CSE acknowledgements section would be complete without mentioning our amazing support system. Lindsay Michimoto keeps students sane. Although Hank is the chair, we all know that Lindsay is in charge. She is an amazing advocate for students. Lindsay is a large part of what makes UW awesome. Melody Kaneno shelters us from bureaucracy, so we can spend more time goofing off. Conveniently, she can also be paid in chocolate.

My relationship with my roommates has been a large part of my Seattle experience. I prefer to think of them less as roommates and more as family. I'd like to thank Ben Birnbaum for being just a little bit sad, a lot a bit awesome, and showing me that success is not a shackle; Jonah Cohen for challenging my beliefs, keeping an open mind, and pushing me to face my fears; Joe Devietti for being willing to do stupid things like stair workouts at 6am on Saturdays; John John for being inhumanly easy going and providing entertainment and perspective; Travis Kriplean for listening, unravelling my stress, and guiltily sleep-eating my almond butter; Yongjoon Lee for late night Taco Bells trips followed by even later night Starcraft matches; and Susumu Harada for hilarious moments of confusion and inspirational moments of kindness. Roommates, in general, have been critical in my development. Much of where I am today is a product of random chance. Gilman Tolle, my freshman year roommate, has left a large imprint on on my personality. I had the luck of rooming with one of the smartest computer science students at Carnegie Mellon for four years. My interest in HCI, my intellectual curiosity, and my echoing laugh are all stolen from Gil.

My support network in Seattle goes extends far beyond my research group and roommates. There are too many folks to credit, and I will inevitably forget some. If I do forget you, I apologize. I still love you; don't be sad.

The networks lab (Mike Piatek, John John, Tomas Isdal, and Dan Halperin) has given me their friendship, an outlet for nerdiness, and a reliable source of lunchtime company. My running and workout groups (Ben Birnbaum, Tomas Isdal, Christoffer Klang, Alex Jaffe, Jo Crook-Barr, Abe Friesen, Yaw Anokwa, Joe Devietti, Travis Kriplean) have transformed me from a scrawny nerd to a scrawny nerd that can run fast and lift heavy things. Not only did the exercise keep me sane, but the workouts also provided me with amazing memories. Running through Wallace Falls is one of my favorite memories from grad school.

Folks don't fit neatly into buckets, sometimes you have to shove and push to really get them to fit. I thought it might be easier to single some folks out. Alexander Jaffe and Nodira Khoussainova have had a six-year close-up view on my evolution as a person through grad school. There are large spans of time where I saw them every single day. I'm sure they're sick of me by now. Through the years, they have seen me through a number of broken hearts, helped me cope with crushing failure, and celebrated my successes. They have kept me grounded, and without them this process would have been far more difficult. Jennifer Klein has become an extension of my biological family, bravely attending two Indian weddings. Her open ear and logical mind have changed the way I live my life. Latika Kirtane has acted as a proxy for Amee, my sister. She has tied my rakhis and provided me with a limitless well of positive energy. Mike Piatek has provided a shining example of how to fight momentum and make decisions based on happiness. Yaw Anokwa has committed treason and dragged a severely intoxicated version of myself through the streets of Berlin all in one night. In a sea of couples or single men who have given up, Abe Friesen has been an invaluable resource for discussing single guy issues. Elizabeth Tseng has inspired me with her broad talents. Ian Simon has lent me his infectious idealism and love for crosswords. Rishi Sanyal has inspired me with his intense, and at times, insane, obsession with his art. Shaheen Gandhi has shared his passion for making cool things that people use. Jo Crook-Barr has motivated me to be a better person with her frightening, yet lady-like, physical strength, strong sense of purpose, and laser-like focus. Eva Ringstrom has provided a steady source of delicious food and delightful company. Paul Pham has shared a common struggle with balancing western and eastern values and has tamed large elephantine vehicles. Emily Jacobson and Alexis Hope have been a firehose of wonderfully uncomfortable situations and willfully awkward moments. Justine Sherry has enriched my life by living in my computer, reading me bedtime stories, and just plain telling me nice things. Drew Perdichizzi has seen to it that my body doesn't fall part. Michael Bernstein has been a constant source of support through this painful job hunt process.

I'd like to thank everyone that went out of their way to help me during this long thesis writing process. Thanks to Didi Isdal for calming me down, making me dictate parts of my thesis, and transcribing what I said. Thanks to Yaw Anokwa, Seema Patel, Amy Ferris Wheelis, Gilman Tolle, KC Oh, Elisa Celis, Danielle Kulp, Rishi Sanyal, and Sam Kaufman for debugging my writing. Amy actually read my entire thesis, which makes her the first non-advisor to read it. Hopefully it wasn't too scarring.

Of course, where would I be without the Seattle coffee shops. I have spent most of my grad life "working" in cafes. I will miss that culture dearly. Thank you to Remedy Teas, Pioneer Coffee, Victrola, Trabant, Milstead, and Zoka.

None of this research would be possible without generous funding from the American taxpayer. I'd like to thank the National Science Foundation for funding through grants IIS-0812590 and OAI-1028195 and the Army Research Office for funding through the NDSEG fellowship. I've also been the beneficiary of generous corporate support in the form of gifts from Google and Microsoft and also the Microsoft Graduate Research fellowship.

Finally, being in grad school is a incredibly selfish act. You can, if you choose, remove yourself from the concerns of the real world and work on problems you find interesting, for no other reason than because *you* find them interesting. The motivations behind such a life choice and the consequences of it can be hard for many families to understand, especially Indian families. My parents and sister have been extremely supportive. Without their support and guidance none of this would be possible. Thanks Mom, Dad, and Amee!

Chapter 1 | INTRODUCTION

Computers are *everywhere*. Desktops are on most every desktop. Computers keep track of critical systems in cars and airplanes. They connect us to the world through our cell phones and tablets. When we come home, looking for entertainment, we use the computers in our televisions and video game consoles. But computers by themselves are just machines that process instructions. For a computer to do anything it needs instructions to execute. It needs a program.

A program instructs a computer how to behave, how to process instructions. For this dissertation, I focus on a specific programming task: writing a function. A function maps inputs to outputs. Programming is the process of specifying this function.

One way of programming a computer is by **authoring a function** through code. When authoring behavior, programmers design a function and write code to provide a computer with the exact instructions on how to map inputs to outputs. Consider the scenario where a programmer is trying to author a function that sorts a list of names. There are many ways they could do this. For example, they might manually specify the operations needed to reorder the list of items. Or they might find a library that contains a function that sorts the list. In either case, the programmer can reason about the preconditions and postconditions of the function. They know what will happen when they apply their sorting program to a list. If the function doesn't behave as expected, it's because there is a bug in the code. Either the programmer or the creator of the library didn't describe the function correctly.

Authoring behavior is the most common way to program a computer. And it's important: there are encyclopedias worth of code powering widely used tools like Microsoft Office and Adobe Creative Suite. As a consequence, research and industry have focused on understanding and supporting programmers as they write, edit, debug, and share code. Tools such as integrated development environments, code versioning systems, and debuggers have lowered the barrier to programming.

However, authoring behavior is only one way of specifying a function. There are useful functions that can't easily be specified manually through code. Consider the scenario where a programmer wants to create a function that takes speech input, recognizes that input, and outputs text. Speech input varies and is greatly influenced by outside factors (e.g., the speaker, recording equipment, recording environment). Programmers can't write robust code to account for all of these variations, but they can observe the input space to collect examples of speech. They can leverage their knowledge about what aspects of speech input might be used to distinguish between inputs and encode that knowledge in a form a computer can understand. They may even be able to provide sample outputs, in the form of corresponding text transcripts, for each input they observe. The computer can use all of this information to learn its own function.

To create these functions, programmers and computers need to work together. The computer needs the programmer to collect observations and write code to provide descriptive information about those observations, and the programmer needs the computer to use this information to find patterns. In the literature, observations are referred to as data and functions built from data are called models. In contrast to functions created by programmers authoring behavior through code, these functions are created by programmers **training a model** and computers **learning a model** through data and code.

When authoring behavior, programmers often have a design specification in mind. This specification outlines the desired behavior that is possible to achieve with the right code. If the function doesn't behave as desired, programmers can change the code to fix the behavior. Training a model introduces new challenges. These challenges stem from the fact that the behavior of the model is not solely determined by the code – it is determined by both the code and the data. If the learned function performs poorly, it might be that the data, algorithms, and computational resources are not sufficient to solve the problem. For example, in speech recognition, state-of-the-art systems still fail when applied to difficult data.

These challenges change the way functions are programmed. Unlike authoring, where a programmer can simulate behavior in their head or trust a black box implementation from a library, training involves running experiments to understand what the computer has learned. Programmers experiment by changing data and code to see if it is even possible to build a function that behaves the way they want it to.

Little work has been done to understand the programming process for training models, and tool support for training a model lags far behind support for authoring a function. In this dissertation, I study the programming process for training models, focusing on the use of a specific set of techniques known as machine learning algorithms. I refer to the programming process for training models using machine learning as the **machine learning process**.

In the next section, I describe machine learning and focus my discussion on a specific set of algorithms called classification algorithms. Programmers train models using classification algorithms by providing examples of inputs and outputs. I then provide a scenario in which a programmer uses classification algorithms to train a model that recognizes handwritten digits. I use this scenario to explore the differences between *authoring a function* with code and *training a model* with code and data. I then present my thesis statement and provide an overview of my dissertation, describing at a high level the studies I have run and the tools I have built to support my thesis.

1.1 Machine Learning

Machine learning is a discipline that focuses on programming functions by showing them how to behave through examples of data. A machine learning algorithm describes a computational process through which a computer models data. *The learned model is a function from inputs to produces outputs.* Tom Mitchell provides a well-posed definition of what it means for a computer to learn [96]. He says: "A computer program is said to *learn* from experience E with respect to some class of tasks T and performance metric P, if its performance at tasks in T, as measured by P, improves with experience E."

To train behavior, a programmer first conceives a *task* that they want the computer to solve.

For example, the programmer may want the computer to recognize speech and transcribe it to text. They then provide the computer with examples of *experiences*. These examples are commonly called data. Data is an observed proxy for a behavior related to the task. For the speech transcription task a programmer would collect audio files containing speech with associated transcripts. The data shows the computer what it should be doing. Machine learning algorithms detect patterns in the data to build a model. The usefulness of that model is assessed through a *performance metric*. In the speech transcription example this metric would be how well the system recognizes speech.

Programmers must concretely define experiences, tasks, and performance metrics for their problem. The way a programmer defines a problem determines which algorithm they should use. For example, programmers typically solve speech transcription problems using supervised learning algorithms. The programmer provides the supervised algorithm with inputs in the form of examples and supervision in the form of outputs. Supervision itself can be subdivided into discrete outputs (i.e., classification) and continuous outputs (i.e., regression).

There are problems for which the programmer may not know the outputs ahead of time or it may just be difficult to generate credible outputs. For these problems programmers are looking for patterns – if certain examples group together. For instance, a programmer working with user activity logs may want to group similar users together to understand their usage patterns. These problems are known as unsupervised learning problems because programmers are not providing any additional information, or supervision, about the output. These problems are commonly addressed using clustering algorithms.

The space of machine learning algorithms expands past classification, regression, and clustering. Reinforcement learning algorithms rely on a different representation in which programs perform actions in the world and evaluate the consequences of those actions to learn the right model. Hybrid techniques combine aspects of different algorithms. Machine learning itself provides a rich set of different approaches for training models.

In this dissertation, I focus on classification algorithms. Training behavior through classification is straightforward. The programmer provides examples of both inputs and outputs to the function, and the computer learns a mapping between inputs and outputs. This technique is incredibly powerful. Companies like Google, Yahoo, and Microsoft use classification to learn models of spam and non-spam email to remove spam emails from inboxes [116]. Biology researchers use it to recognize patterns in the human genome [37]. And the U.S. Postal Service (USPS) uses it to route physical mail by automatically recognizing the handwritten address on an envelope [85]. To understand the process of training a model, we can take a closer look at the last scenario, handwriting recognition.

1.2 Scenario: Routing Physical Mail

Hundreds of millions of boxes, envelopes, and postcards move through USPS sorting centers each year. Each piece of mail needs to get to a destination address. Before 1994 routing mail was a manual process: a human had to inspect the handwriting on each package. The automation of this process is noted as a huge success for machine learning, and has greatly increased the efficiency of the postal service.

The systems that the USPS uses today recognize the entire address, but initial systems focused on the simpler problem of recognizing a handwritten zip code. Each address can be geographically localized by looking at its zip code. This five to nine digit number helps the USPS route mail efficiently. Looking at how a handwritten digit recognition system is built provides insight into differences between training behavior and authoring behavior.

Figure 1.1 shows the entire handwritten digit recognition system. Training a model is just a small part of the overall system, but it is influenced by the needs and constraints of the system. The effectiveness of the classifier also influences how the system is designed. To ground the new concerns raised by training a classifier and the dependencies between the model and the rest of the system, let's observe the steps Ada takes to train a handwritten digit classifier.

Gather Data and Define the Problem

Training a model requires both code and data. Before Ada can use a classification algorithm to learn a function, she must have examples of the inputs and outputs. The first step is to *create a digital representation* for physical artifacts. Ada digitizes mail by scanning the handwritten address and converting it into a image file. Next she needs to find the zip code in the image. Finding the zip code can be a difficult machine learning problem itself, but for the purpose of this scenario let's assume Ada has such a system. At the end of this process Ada will be left with a digital representation of the zip code for each piece of mail.

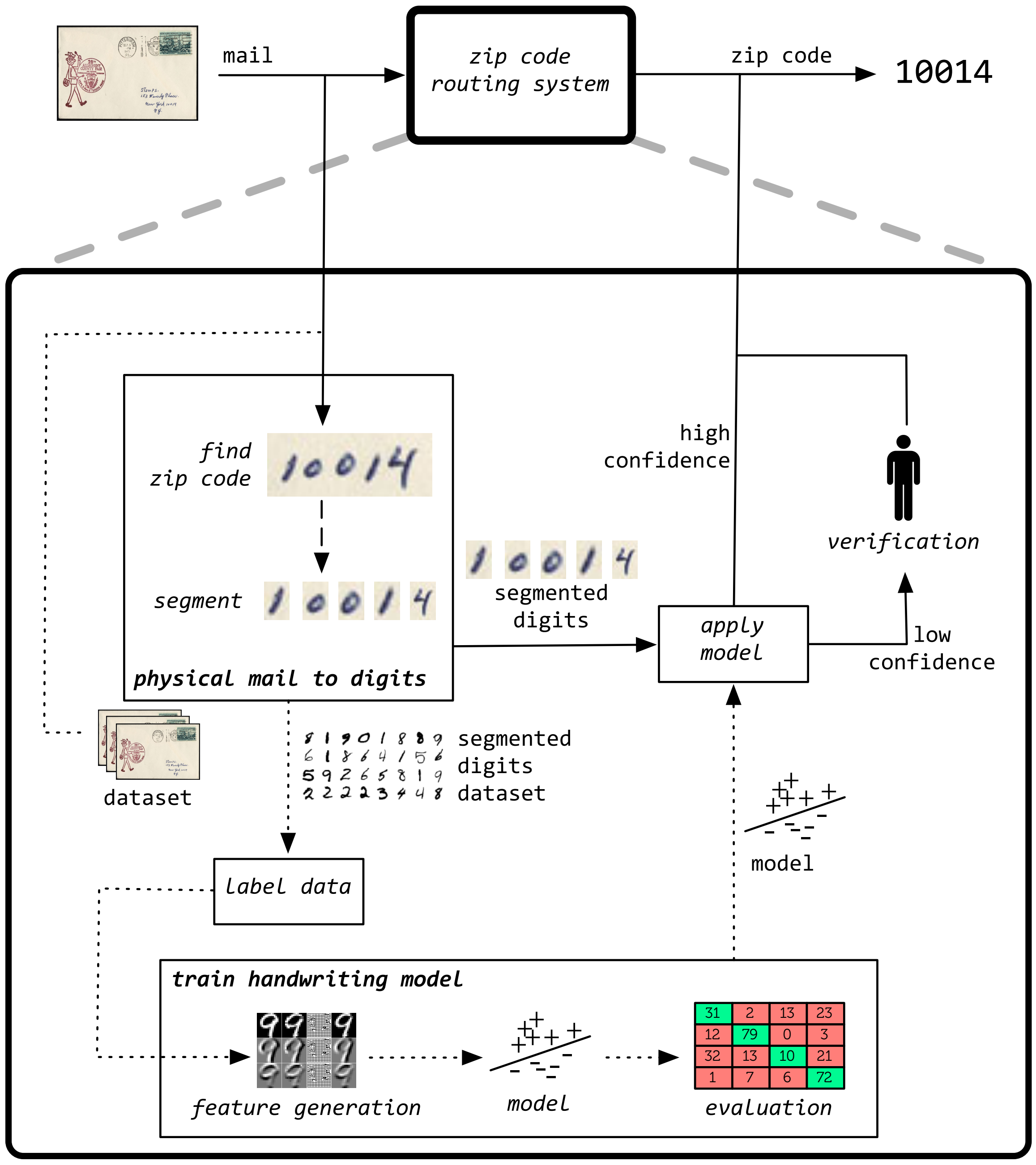


Figure 1.1: Training a model is just part of the process of building a handwritten digit recognition system. The dashed lines show the steps involved in training a model. The solid lines show the steps involved in applying the model. Note there is overlap in transforming physical mail to digits.

Ada must now clearly *define the problem* she wants the classifier to solve. This can be a complex choice. For instance, Ada may choose to recognize the entire zip code, or she may choose to further subdivide the problem by segmenting the zip code into component digits. Segmentation is another processing step, but in practice it reduces the complexity of the classifier and produces robust results. Ada segments her image, and the classification problem is now reduced to recognizing a single handwritten digit.

Ada now has a function that gets digital images from physical mail. But she does not yet have a dataset which she can use to train a model. To *create a dataset* Ada uses her system to scan thousands of envelopes to gather tens of thousands of digit images. Since classification is a supervised technique, Ada provides an output for each input. This means she must provide the ground truth number for each digit image in her dataset. Generating ground truth outputs for inputs is called *labeling data*, and it often requires human labor. Ada labels her dataset by paying people to look at digit images and label each image with its corresponding digit.

Generate Features

After labeling data, Ada finally has the set of inputs and outputs she needs to train a model. However, the inputs are not yet in a form that most classification algorithms can understand. Ada needs to describe image attributes that can be used to distinguish between different digits. This process is called *feature generation*. Features are numerical, nominal, or relational values used by the classifier to find patterns in data.

Train and Evaluate a Model

After Ada has generated features, she can *train a model* using a classification algorithm. She needs a metric to understand how *evaluate the model*. Ada chooses to evaluate the digit recognition model by measuring accuracy. She splits the dataset into testing and training sets. The training set will be used to train a model, and the testing set will be use to see how well the model works. To simulate the effectiveness of the model on new data, Ada ensures that the training and testing sets do not overlap. After she has set up this experimental framework, Ada can try a variety of different classification algorithms and compare their accuracies.

Test and Iterate
Ada finds that her first attempts to build a model don't work well; the models have low accuracy. In the process of *improving performance* Ada generates new descriptive features. She analyzes her data to make sure it is a good sample of handwritten digits and collects more data to account for any discrepancies. She debugs her machine learning code to remove errors in the implementation of the algorithm.

Once she is happy with the accuracy, she plugs her handwritten digit recognition model into the larger automated mail routing system. After interacting with the production system, Ada realizes she will not be able to build a perfect classification system – handwriting data in the real world is too varied. This means there will be instances where the classifier will fail, and the real system needs to be able to handle those cases. She modifies the model to provide confidence measurements for each prediction, and she modifies the routing system to add an extra human verification step for predictions with low confidence.

The construction of the handwritten digit recognition system is more complicated than just choosing an algorithm and training a model. Ada makes a number of decisions along the way, decisions about how to collect, process, and model data as well as how to test the model and account for prediction errors real production systems. These decisions not only effect the accuracy of the model, but they also affect the design of the final mail routing system. However, there is structure in the programming process. By looking at how Ada programs the handwritten digit recognition system, we can better understand and support training behavior through data.

1.3 The Programming Process

The machine learning process is different than coding. The key difference is the addition of data as a key ingredient in defining how a trained function processes information. This means that if the handwriting recognition model is working poorly, the source code may be flawless and the problem could likely be solved by providing better data.

Since the behavior of the model is based on data, all of the steps involved with gathering and processing the data affect the behavior of the program. There are a number of different steps, and each step can have errors. In the handwritten digit recognition scenario above, the transition from physical mail to labeled is a multi-stage process. Ada gathers digital images of addresses from physical mail, localizes zip codes, segments zip codes into digits, and labels digits. Additionally, Ada must ensure that her training data is representative of the data her function will encounter in the real world. She also needs to make sure that her testing data is separated from her training data to simulate real world distributions of data. Errors in any of these decisions have downstream effects on the performance of the model.

Code is as important as data. Programmers still need to write code. In the handwritten digit example, Ada writes code to localize and segment zip codes, generate features from data, train a model using a classification algorithm, and run and track experiments. Interactions between the data and code can lead to bugs in non-obvious locations. For example, address images gathered with slightly different cameras may reveal a bug in zip code segmentation. Ada must be willing and able to revisit all of the code.

Given the complexities and error prone nature of the process one might wonder – why even bother with machine learning algorithms? Ultimately Ada has no other choice; a machine learning approach is the only realistic way of dealing with the complexity of handwritten digit recognition. Due to the sheer number of different ways to write digits, Ada cannot manually build an effective model. And because certain digits can often look very similar (e.g., fours and nines) the computer cannot easily classify the data with accurately. The fact that neither the computer or Ada have a complete and correct model means they have to communicate their partial knowledge of the model to each other in order to train a good model.

Communication between the programmer and the computer happens through many different channels. The programmer communicates through data and code, and the computer communicates by providing a model and experimental results. In the example above, Ada communicates inputs and outputs through labeled digits, how to tell digits apart through feature generation code, how to build a model through classification algorithm code, and how to tell if a function is good enough through code that splits the data into train and test sets and evaluates the performance. The computer communicates what it has learned by providing a model and predicted labels for digits in the test set.

Ada tries to improve the function through experimentation. She changes the data and code to see if she can improve the performance based on her current understanding of the learned model's limitations based on previous predictions. The computer provides new predictions based on those changes. For example, Ada provides new digit data, and the computer provides an increased accuracy score letting her know that the new data helped it train a more accurate model. It is through this process of experimentation, that both Ada and the computer build a better model of the data.

The machine learning process is different than the process for authoring a function. Algorithms can no longer be treated as black boxes, and programmers can't be expected to model the problem in their head. It is a process defined by code, data, and experimentation. It is a data-oriented programming process. The bulk of this dissertation focuses on understanding and supporting the machine learning process.

My studies show that many experienced programmers have a hard time using machine learning algorithms because they are not familiar with the machine learning process. Current general-purpose programming current tools are not designed to support this process. Integrated development environments (IDEs) support writing, debugging, and managing code, but provide little to no support for data. Current machine learning support for programmers still focuses mostly on the coding aspect of the process. For example, machine learning APIs reduce the coding burden by providing libraries of algorithms but do not help the programmer manage data [137].

My thesis focuses on supporting machine learning programming through better general-purpose programming tools. I do this by building tools that support the machine learning process, a programming process that uses both code and data to create a program. To this end, I have run studies and created new development tools in support of my thesis:

> **The programming process of using machine learning to train a model is different than the programming process of authoring a function through code and is poorly supported by current general-purpose programming tools. We can better support machine learning though new programming tools that support writing code and understanding data.**

1.4 Outline

The dissertation will be organized as follows:

Chapter 2 describes related work. Machine learning programming involves writing code and providing data to train a function and conducting experiments to see how well the function works. I cover related work that supports writing code, working with data, and conducting experiments. I also cover current machine learning tools and compare them to my approach.

Chapter 3 describes my initial studies of machine learning programming [110]. I describe and provide results from two studies. In the first study, I interviewed researchers applying a variety of machine learning algorithms to a diverse set of domains. From these interviews I extracted a general programming process for machine learning that I call the machine learning process. I ran a second study to observe the machine learning process in a laboratory study. I use these two studies to describe the machine learning process and how current programming tools fail to support this process. I focus my development support on a specific subset of machine learning, classification.

First, I explore how a new programming environment designed around the machine learning process can better support classification. Chapter 4 presents Gestalt [108]. Gestalt helps programmers train classifiers by connecting code to data. Gestalt supports organizing, editing, and writing code. Programmers can interact with data by visualizing datasets and looking at connections between classification results and examples in the dataset. My studies show that programmers are better at debugging a trained function with Gestalt than with a state-of-the-art general-purpose programming tool.

I build on my work on Gestalt, by looking at how new analysis tools can help programmers understand limitations of their datasets. Chapter 5 describes Prospect [109]. Prospect automatically generates many different configurations of features, classification algorithms, and evaluation techniques. These configurations are used to train multiple models for the same dataset. Each model provides a predicted value for an example in the dataset. Agreement between models is used to generate new interactive visualizations, which in turn help programmers better explore and understand their dataset. I show how these visualizations can be used to help programmers find noisy examples in their dataset and create more descriptive features.

Finally, I look at how new programming tools can support experimentation, which is a key part of the machine learning process. Chapter 6 describes Hindsight. As programmers change parameters and datasets they leave an experimental trail. Tracking these changes is an important but difficult task. Hindsight uses the structure of a classification problem to automatically track changes and associate them with results. It allows programmers to compare experiments to better understand how changes in parameters and code change results.

I close by providing examples of how this work might be extended and conclude my dissertation in Chapter 7.

Chapter 2 | RELATED WORK

Training a model can be broken down into three high-level tasks. First, programmers write and manage code to create a data flow. Second, they collect, store, and analyze data. This data provides learning algorithms with training examples, which are used to train a model. The data also helps programmers understand the behavior of the model. Third, programmers run experiments by changing code and data to explore the space of possible models. The purpose of this experimentation is to find a particular model that works well for their problem.

There is a rich history of related work for each of these tasks. Researchers have studied programmers and have built tools for writing and understanding code, analyzing data, and supporting experimentation. The first three sections of this chapter will survey the wide range of work in these three areas, and will compare existing solutions to my own work. The last section will survey tools that have been specifically designed to support machine learning and will contrast those tools with the tools that I have built.

2.1 Writing and Managing Code

In order to train a model, a programmer has to create a data flow. This flow describes the computational transformations that take raw data and turn it into a model. The flow can consist of multiple steps: parsing data files, fusing different data sources together to create a dataset, extracting features from the dataset, and training a function based on those features. As is true in any other programming task, programmers will make mistakes when creating this data flow. These mistakes lead to bugs in the control flow of the code (i.e., incorrect instructions executed by the computer). Understanding the control flow is important for fixing these bugs. In this section, I describe related work on creating, managing, and understanding a data flow.

2.1.1 Creating a Data Flow

The high level control flow of many programs can be represented as a data flow. A data flow is a graph where each node is a piece of computation, and data flows along the edges. Sutherland provides an early example of how such a system might work [128]. Sutherland"s system allows programmers to enter arithmetic instructions using two different representations: textual code and visual data flow graphs. Both representations lead to the same result, but the data flow representation decomposes the problem into a form that is easier to interpret.

Interpretability is just one of many reasons why research and industry have exhibited continued interest in data flow programming [68, 139]. Because the computational flow is explicitly represented as a graph, programmers can use graph analysis algorithms to find independent components and execute those components in parallel for improved performance. Data flow representations also provide built-in modularity, because the graph structure allows programmers to easily swap out components that have the same types.

Researchers have frequently exploited the visual nature of the data flow to build better programming tools. Visual languages derived from data flow development are popular in the end-user programming community, where they are used to make programming accessible to people without experience writing textual code [139]. Modern programming tools like Visual Studio [7] and Matlab [10] include some support for visual data flow programming. Prograph provides a vision of what a general-purpose visual programming language might look like [36].

Data flow programming tools are effective for programming tasks where the workflow has a natural graph structure, such as science and engineering. Scientific workflow systems such as Taverna [105], VisTrails [30, 31, 119, 120], and Kepler [89] provide support for a variety of data intensive tasks in a variety of scientific domains (e.g., gene-alignment in bioinfomatics, wild-life tracking in ecology). Such workflow tools have seen commercial success in systems such as LabView [25]. Data flow programming tools like Rapid Miner [93] and Knime [24] extend this type of support to machine learning.

My tools build on top of this work. Gestalt and Hindsight are both general-purpose development environments that support the construction of a data flow pipeline. Unlike Prograph, the graphical structure in Gestalt and Hindsight is not meant to replace coding. Programmers still need to write code. Rather, like LabView, I take a hybrid approach by allowing programmers to mix both imperative and graphical programming. Unlike LabView, which looks at the data flow as the key programming component, my tools look as the data flow as an organization aid – a rough scaffold upon which code can hang. This distinction is important. Gestalt and Hindsight place programming front-and-center to prevent the data flow from getting in the programmer's way, which ensures that the tools remain flexible enough to address new problem domains [100]. The data flow scaffolding allows these tools to support programmers who are training a domain-specific model, while still taking advantage of the improved modularity provided by a data flow language.

2.1.2 Debugging a Control Flow

Program errors are as old as programming itself. Ada Lovelace, regarded by many to be the first programmer, reflects on the process of programming when describing Charles Babbage's revolutionary computing machine, the Analytical Engine:

> an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.

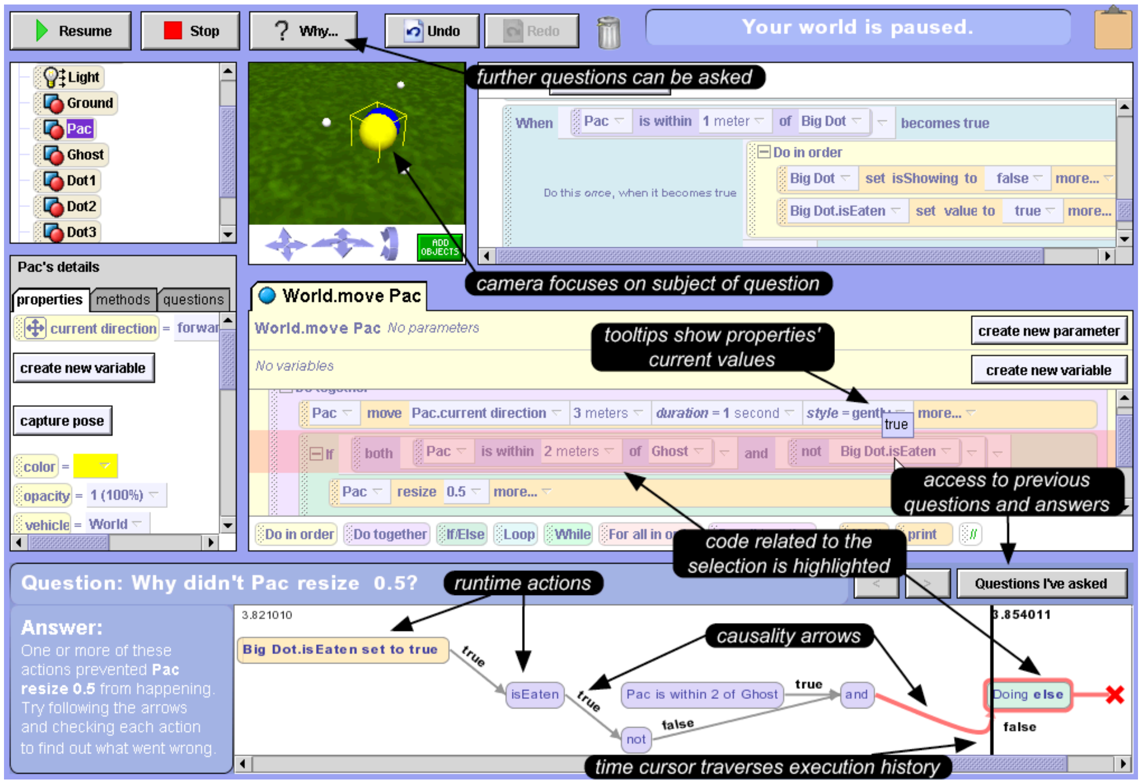
Ada Lovelace recognizes that the computer is working properly – it is properly executing instructions provided by the programmer. Rather than the machine, the errors are result of poorly constructed "orders" or bad code [88].

As programming matured, new techniques were created to help programmers debug code by understanding the control flow of a program. Techniques like assertions allow programmers to place a loose contract between the inputs and outputs of functions as they write code [60]. Programmers can assert preconditions and postconditions to functions, and the program will halt with an error if those assertions fail during runtime. Printing values to the console allows programmers to debug the control flow by tracing the execution path of the program and logging relevant information. Debuggers allow programmers to control the execution of a program [113]. Programmers can stop the program at specific breakpoints and inspect its state. Graphical user interface debuggers make this task even easier. These tools allow programmers to specify breakpoints by clicking on a line of code and using interface widgets to view relevant state, such as active variables and the execution stack.

Understanding how programmers make mistakes can inspire new interfaces that help them avoid errors. This is a key motivation for programming studies, and in particular, studies of programming errors. In *The Errors of TeX*, Don Knuth reviews and categorizes all of the programming errors and enhancements he made while creating the TeX layout engine [74]. Knuth concluded that knowing about what kinds of errors he makes didn't help him avoid future errors. To Knuth, making errors was part of the process. Making these errors provided him with experience and helped him better understand his program. In undertaking this study, Knuth learned that he will always make errors and that he must "strive energetically to find errors in [his] work." Errors will always exist, but tool support can make finding errors easier and less painful.

Knuth's study is one of many studies that seek to understand programming errors. Ko and Meyers survey these studies and develop a framework for studying both the programmer and the programming system together [75, 77]. Based on this framework, Ko and Meyers develop a methodology for studying programming breakdowns. They use this methodology and framework to study the Alice programming system. This study showed that programmers systematically form bad hypotheses about why the system is behaving the way it does, and found that these bad hypotheses were at the root of 50% of the errors observed in their study of Alice.

These studies led to new tools, such as the Whyline for Alice [76]. The Whyline allows programmers to ask "why did?" or "why didn't?" questions about their program in order to understand its behavior. Figure 2.1 shows an example of a question asked by programmer trying to understand why their 3D object didn't change size. In the Whyline, programmers formulate questions by selecting options from a drop-down menu. These questions are grounded in the output, inducing programmers to ask questions when the output doesn't match their expectations. In a user study, programmers found that output-based question asking was both useful and intuitive, and they were able to greatly reduce debugging time when using the Whyline. The Whyline has been extended to support end-user debugging in deployed applications [102], in Java development [78], and in end-user debugging of



```
When  Pac is within 1 meter of Big Dot becomes true
  Do this once, when it becomes true
    Do in order
      Big Dot set isShowing to false more...
      Big Dot.isEaten set value to true more...
```

World.move Pac

World.move Pac *No parameters*

No variables

```
Do together
  Pac move Pac.current direction 3 meters duration = 1 second style = gently more...
  If both Pac is within 2 meters of Ghost and not Big Dot.isEaten
    Pac resize 0.5 more...
```

current direction = forward

color =

opacity = 1 (100%)

vehicle = World

true

Question: Why didn't Pac resize 0.5?

Answer: One or more of these actions prevented **Pac resize 0.5** from happening. Try following the arrows and checking each action to find out what went wrong.

3.821010 — Big Dot.isEaten set to true → true → isEaten → true → not → false → and; Pac is within 2 of Ghost → true → and → Doing else → false — 3.854011

Figure 2.1: The image above shows the result of asking a question about the program's behavior in the Whyline. The question is displayed below. The Whyline highlights both the code and the objects in the 3D environment related to the question. It also visualizes the set of instructions that led to the current state. This visualization provides the programmer with a straightforward way of backtracking to find the cause of the error.

deployed models [82].

While useful, the extension of Whyline to machine learning targets a different problem than I do [82]. Whyline is targeted towards end-users debugging deployed models in applications and is constrained to fit the algorithm decisions for a specific system in the context of a specific problem domain. The Whyline doesn't provide general-purpose support for training a model. I discuss differences between my tools and existing tools for machine learning in more detail in Section 2.4.1.

Although I do not strictly follow the methodology prescribed by Ko, my studies and tools are influenced by his programming studies [75, 77] and his work on Whyline [76, 78]. In my studies, I construct a similar task, designed to observe programmers as they train a model with current tools. This study reveals problems with existing tool support, including problems debugging trained models when the behavior of the model does not match expectations. These problems inform my subsequent work on general-purpose tools for training a model using machine learning.

I also draw inspiration from Ko's studies of programming barriers experienced by novice programmers using traditional programming tools [79]. In this study he discusses barriers faced by novice programmers when they are learning how to program. One such barrier described by Ko is an understanding barrier. Understanding barriers occurred when a "novice could not evaluate the program's behavior relative to expectations." Since they didn't understand why the program behaved the way it did, novice programmers could not fix errors.

Because most programmers are novices when it comes to using training a model, there are parallels between novice programmers trying to author a function and novice machine learning users trying to train a model. For example, programmers also experience understanding barriers when their trained model performs poorly. My studies show that most programmers don't have the mathematical background needed to understand how a machine learning algorithm converts data into a model.

Programmers need new tools for understanding the behavior of trained models. Authored functions have few data dependencies. This means that that it is feasible for the programmer to interact with dependency chains. Tools like the Whyline can exploit the fact that there are few data dependencies in authored programs. For example, the program in Figure 2.1 only has four data dependencies. In contrast, trained models are dependent on large datasets; they have thousands of data dependencies. Programmers can't directly understand all of these dependencies, so they must run experiments to focus their attention on the most important dependencies (e.g., misclassified examples). My tools support programmers by supporting experimentation and highlighting the specific data dependencies that are the most valuable to inspect.

2.2 Analyzing Data

The behavior of a trained model is based on both code and data. If a programmer provides a machine learning algorithm with bad data, the algorithm will produce a bad model. Debugging a trained model involves debugging data, and debugging data involves understanding the dataset being used to train the system, the learned model, and the experimental results. In order to understand the relationships between data and code, programmers make heavy use of visualizations, charts, and graphs. Researchers have studied how people understand data and have built tools that help people better understand data through visualization and analysis. In this section, I cover related work in three areas: sensemaking and interactive visualizations, programming support for visualization, and visualization support for specific machine learning algorithms.

2.2.1 Sensemaking and Interactive Visualizations

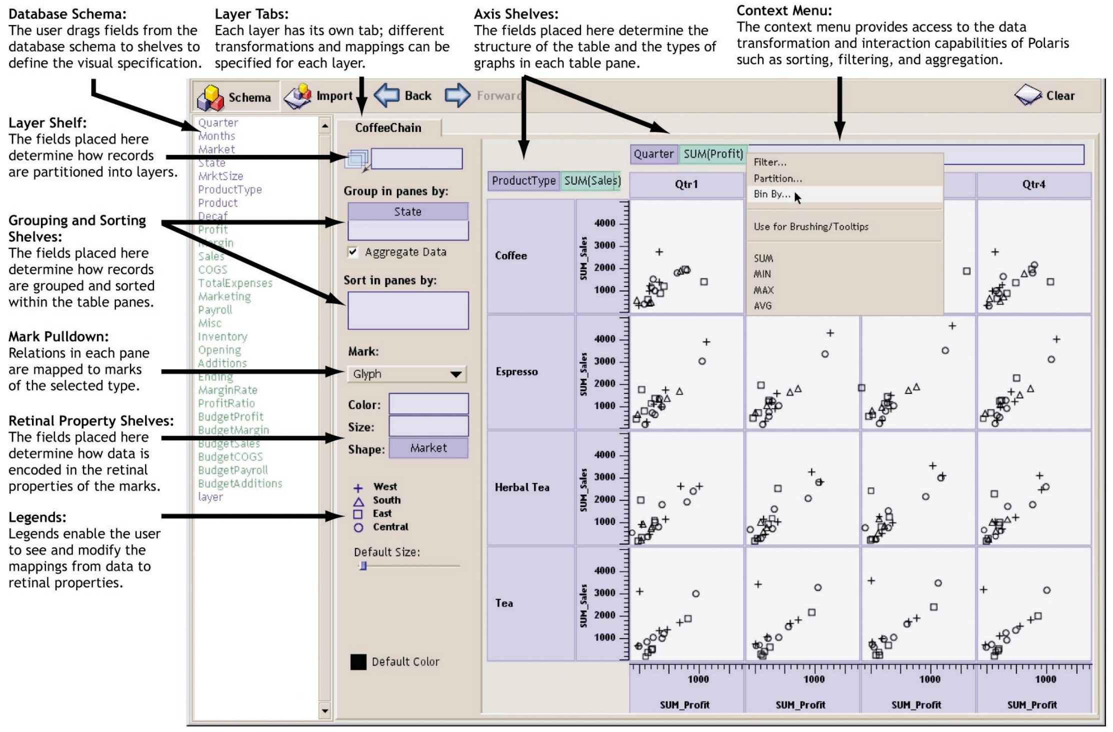
Sensemaking is the process by which people understand data [39]. In the human-computer interaction literature, Russell et al. frame sensemaking as a learning loop in which people search for representations of their data, instantiate those representations, and shift representations based on new data [115]. People then use the generated representations to solve a problem they are working on. Russell provides a framework for analyzing the cost structure of this process. In this framework, visualization tools support users by reducing the cost of common actions. Reducing costs increases the efficiency of the current users and lowers the barrier for new users.

Researchers have designed new interactions that allow people to quickly and easily shift between data representations in order to explore and understand data. Early work by Becker shows how direction manipulation, specifically brushing over variables with a mouse, can help people interactively highlight portions of a visualization [23]. Interactive highlighting helps focus attention on specific portions of the data. It also helps programmers easily switch between portions of interest to discover patterns and make sense of data. Visualization tools evolved to include new interaction techniques such as dynamic queries through faceted search, which allow programmers to sort, filter, and color data based on variables in the dataset [13, 121, 122, 136]. More recent work on collaborative interactive visualization tools looks at how tools can facilitate sensemaking in group settings [57, 59].

Most early interactive visualization work focused on specific domains [13, 136]. Over time, researchers extracted general principles for building interactive visualization tools [123]. Shneiderman identifies an effective process for exploring data: "Overview first, zoom and filter, then details-on-demand." First the system should provide a high-level overview of the dataset containing only the most important information. Users should then be able to dig into data by zooming into portions of the overview visualization and/or filtering out examples. Zoom and filter allows users to push away parts of the dataset they don't care about so they can focus on subsets of interest. Finally, the system should provide details when users ask for details. Shneiderman elaborates on this process, on datatypes that might need to be visualized, and on the actions that users take when exploring data though interactive visualizations.

Current general-purpose visualization tools build on these abstractions. Because machine learning data is typically stored in a data table, the tools most relevant to machine learning programming are general-purpose database visualization tools. Research projects such as Polaris allows users to load and visualize structured data [126]. Figure 2.2 shows the Polaris interface. Users connect Polaris to a structured data source (e.g., spreadsheet or database) that they want to visualize, and Polaris allows users to create charts and graphs by interactively specifying which columns they want to compare. Polaris recognizes the data type of each column, and based on the type it selects appropriate visualizations. Commercial tools like Tableau [12] and Spotfire [11] build upon Polaris to provide mass market general-purpose visualization support.

The visualization components of my tools are heavily influenced by prior work on interactive visualization systems. The analysis interface in Gestalt provides a subset of Polaris's functionality. Hindsight and Prospect provide faceted browsing capabilities for exploring data. Programmers can sort, filter, and color datasets to understand the output of their classifier by zooming in on examples of interest (e.g., misclassified examples). My tools extend current work by providing specific functionality for machine learning. For instance, all



Database Schema:
The user drags fields from the database schema to shelves to define the visual specification.

Layer Tabs:
Each layer has its own tab; different transformations and mappings can be specified for each layer.

Axis Shelves:
The fields placed here determine the structure of the table and the types of graphs in each table pane.

Context Menu:
The context menu provides access to the data transformation and interaction capabilities of Polaris such as sorting, filtering, and aggregation.

Layer Shelf:
The fields placed here determine how records are partitioned into layers.

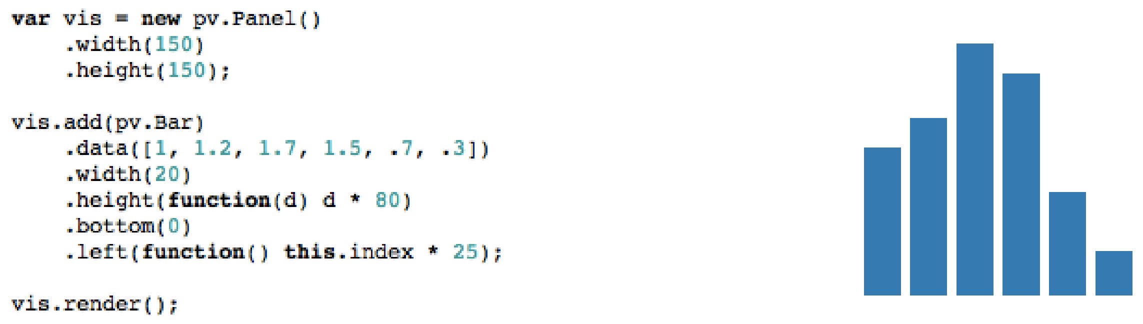
Grouping and Sorting Shelves:
The fields placed here determine how records are grouped and sorted within the table panes.

Mark Pulldown:
Relations in each pane are mapped to marks of the selected type.

Retinal Property Shelves:
The fields placed here determine how data is encoded in the retinal properties of the marks.

Legends:
Legends enable the user to see and modify the mappings from data to retinal properties.

Figure 2.2: Polaris allows users to interactively visualize a database. Polaris reads in a connected database, understands the column types, and provides an interface for creating visualizations by selecting columns.



```
var vis = new pv.Panel()
    .width(150)
    .height(150);

vis.add(pv.Bar)
    .data([1, 1.2, 1.7, 1.5, .7, .3])
    .width(20)
    .height(function(d) d * 80)
    .bottom(0)
    .left(function() this.index * 25);

vis.render();
```

Figure 2.3: Protovis breaks visualizations down into component parts (called marks) and provides API support for composing visualizations based on those parts. This image shows a bar chart composed of bar marks and associated code in Protovis.

three tools provide visualizations that allow programmers to inspect raw examples in their dataset. Linking to raw data is important, since information is often lost when raw data is converted into a structured data source. Programmers need to be aware of the information loss so they can take steps to improve the performance of their model (e.g., writing new features).

2.2.2 Programming Support for Visualization

Database visualization tools reduce the burden of visualizing structured data. However, not all data is structured, and even if it is structured it may not be organized into standard types or there may not be appropriate visualizations. For many tasks, programmers will need to build their own interactive visualization system. Machine learning and visualization share many properties. For example, both have data flow pipelines through which data is processed. Programming support for building interactive visualizations can help inform programming support for training models.

The prefuse visualization toolkit supports the construction of new interactive visualization programs [58]. Prefuse is based on the information visualization reference model, which breaks visualization down into a data flow that loads abstract data, filters that data, and renders filtered data into a visual form. Prefuse provides abstracts that allow programmers to easily construct a data processing/rendering pipeline to visualize data and to create new interactions that allow users to interact with the pipeline and change visualizations.

Tools like Protovis further deconstruct visualizations into their component parts [26].

Programmers compose a new visualization by specifying marks and the properties of those marks. Figure 2.3 shows how a programmer creates a bar chart. The chart is composed of bar marks whose bottom property is 0, and whose height property is connected to underlying data. D3 allows programmers to bind data to a website's document object model (DOM) and apply transformations within the document [27]. By choosing to operate directly on the DOM instead of creating their own data representation, the designers of D3 ensured that their tool can easily interoperate with existing web development tools and programming practices.

All of these tools are built on common abstractions. Heer and Agarwala have encoded these abstractions as design patterns [56]. These patterns provide structure for new programmers trying to build their own interactive visualization tool. For example, Heer and Agarwala recommend a Cascaded Table design pattern for storing data. Unlike row-major tables that make adding new examples fast, cascaded tables make adding new *features* fast. Because datasets are often constant for visualization applications, adding new features is more common than adding new examples.

Abstractions provided by visualization tools can inform the development of machine learning tools. For instance, Hindsight and Gestalt provide a data flow organization similar to the organization described by prefuse. All of my tools store data in a cascaded table, because adding new features to a data table is common as raw data is transformed through the data flow.

2.2.3 Algorithm-Specific Visualizations

If programmers don't understand how a program works, they can't improve it. This is a core difficulty when training a model. Algorithms automatically parameterize a function based on data, and those parameters are used in potentially unknown ways. When the trained model performs poorly, programmers are unable to understand the reason for that failure. Machine learning experts look at learned parameters and use their knowledge of how the algorithms work to make sense of the model's behavior. Researchers have looked at how expert intuitions can be encoded in visualizations to help those without machine learning expertise understand the behavior of their models.

Motivated by providing explanations for results, Becker et al. visualize the internals of a binary Naïve Bayes algorithm [22]. They provide representations that link features to ground

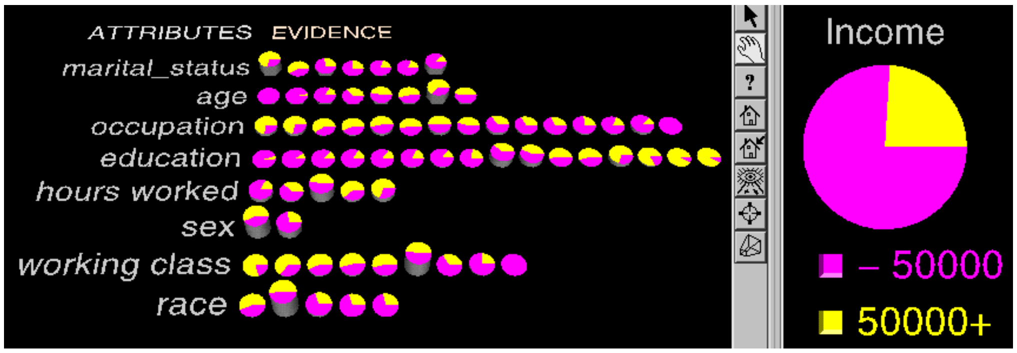


Figure 2.4: A visualization of a Naïve Bayes algorithm by Becker et al. The visualizations show important features, the breakdown by class (pie chart), and the number of instances (height for pie chart).

truth labels and the distribution of examples. For example, their 3D pie chart representation has pies with slices corresponding to the ground truth label and heights determined by the number of examples that have that feature (Figure 2.4). Becker provides features for interacting with visualizations by sorting, filtering, and editing data. Researchers have also provided similar visualizations for other common algorithms, such as support vector machines and decision trees [18, 32].

These visualizations assume some familiarity with the internals of a machine learning algorithm, and that assumption is unrealistic for many programmers. Additionally, the set of algorithms is ever growing. Machine learning researchers are creating new algorithms, often trading interpretability of the algorithm for performance. I focus on general techniques that do not rely on how a specific algorithm works. For example, Prospect works for any classification algorithm. It combines information used to train a classifier (data and ground truth label) with information gained from experimentation (predicted label) to provide useful visualizations of program behavior. My algorithm-agnostic approach makes my tools robust to the introduction of new algorithms, but it does not replace algorithm-specific visualizations. Algorithm-specific visualizations provide a complementary way for understanding the behavior of a trained model.

2.3 Experimentation

Programmers explore the space of potential models by running experiments. They change the code, data, or both, and compare the performance of the new model to the old one. As they experiment, tracking changes and associated results (the experimentation history) becomes increasingly important. Researchers have developed tools in other domains that provide support for capturing history, exploring history, and comparing results. In this section, I look at existing work in these areas.

2.3.1 Tracking Histories

When tasks are wholly within the confines of a computer, tracking experimentation is easy. The computer has the entire program state, and application developers can provide functionality for capturing and restoring previous states. This is the intuition behind the *undo* button [94]. The undo button allows users to try an action without destroying the state. Powering the undo button is the undo stack. The undo stack can be viewed as a historical record of a user's actions – a hidden experimental notebook.

Researchers have long understood the importance of providing users with access to experimentation histories when building applications for exploratory tasks. For instance, when describing how interfaces can enable innovation and creativity, Shneiderman explicitly refers to histories as "central" to improving the quality of a result [124]. Histories are important records of experimentation that can be used to help people reflect on their prior actions and refine their creative work. In addition, user interfaces can use histories to help people interact, explore, and share what they have done to figure out what to do next.

To make use of a history, an application developer first has to capture it. One important decision is when to store application state. For many tools, snapshots are taken automatically every time the application state changes. State changes are often associated with a user action. For some tasks, automatic snapshots do not make sense, because there are too many minor state changes that are not meaningful. In these situations, users may have to manually save snapshots of system state.

For example, when programmers author functions by writing code, the state changes each time they edit a file. These minor edits to files are not meaningful, but capturing history is still important because programmers make errors. Rewinding the code back to a point where the mistake wasn't there helps programmers localize errors [44]. Consequently, modern version control systems are indispensable to most large software organizations.

When using a version control systems, a programmer first creates a repository to store their history and then manually checks in code each time they want to capture a major state change. Version control systems such as RCS [132], CVS [33], and Subversion [35] rely on a centralized repository. Programmers have to be connected to the repository in order to save versions of their system. As connection to a server is often a barrier to use, decentralized systems such as GIT [86] and Mercurial [106] have become popular.

Outside of large organizations with experienced programmers, software management techniques like version control are used less frequently. For example, a recent study of scientists reveals some of the difficulties programmers might have using version control tools. Although they spend over half of their time building software, very few scientists have a good understanding of version control systems. This gap is due to a lack of familiarity with the tools and difficulties setting up version control systems. When scientists realize they need a version management system (e.g., when they need to re-execute an old experiment), it is often too late. These results suggest that automated solutions for keeping software histories can help programmers overcome these barriers.

Automation is difficult with modern version control systems. Unlike previous recording mechanisms where each change corresponds to a user action or a change in the system state, version control systems don't have a good semantic representation of changes. A poor semantic representation means that version control systems don't know *when* to automatically record a change.

Log-structured file systems attempt to solve the *when* problem by keeping track of all changes to files [117]. As storage may become an issue over time, these systems have policies for removing old versions in order to balance a useful historical record with limitations in storage space. In the last few years these systems have seen a rise in popularity, with implementations in modern operating systems [2, 81] and storage services [4].

Even with automated techniques, a lack of semantics is limiting. Log-structured file systems still do not know *what* a change means. Changes in multiple files may correspond to a meaningful change, such as a new feature, or they may not, such as a simple change to a variable name. These changes look the same to the file system. When surfacing version information to a programmer, semantic changes are important for navigating through a large experimentation history.

Recent work has looked into automatically capturing semantically interesting changes. Causality-based versioning [98] looks at the relationships between processes and files to keep track of dependancies between processes and files. Programmers must provide specific guidelines for tasks (e.g., training a model). Once those guidelines are in place, causality-based versioning automatically maintains relationships internally. These relationships are used to intelligently pick *when* to version and to provide rich information about *what* the changes mean.

Once history is captured, the interface needs to expose the history to the user. Modern tools go beyond a simple linear undo and provide rich support for interacting with recorded histories. For example, Amulet provides support for selective undo [101]. With selective undo, users can review the list of actions they have taken and selectively undo actions in a non-linear fashion. Tools like Photoshop [1] and Matlab [5] provide support for selective undo when editing images and writing scripts. Photoshop also provides new interactions, such as the history brush. The history brush allows people to paint changes from a previous canvas onto the current canvas.

Many problems have complex non-linear histories. For large software projects, different programmers explore variations of the code base in parallel. For these projects, version histories have branches, and the structure of the history looks like a tree. Research tools like Designers Outpost have looked at how users can interact with a branched history. Designers Outpost digitizes paper artifacts that are created during the brainstorming process [72, 73]. Like many creative tasks, designers explore and abandon many branches. Designers Outpost automatically captures actions that a designer takes and provides a linear history of the current branch with collapsed portions that reveal other branches. This branch-preserving history allows a designer to go back and revisit prior thoughts when they run into a roadblock. A similar approach to capturing and interacting with branched histories has been taken by commercial data exploration tools, such as Palantir [9].

Capturing and interacting with history is often an essential capability in many pieces of software. In a usage study of Word 2003, undo was the fourth most common used capability, after paste, save, and copy [67]. When researchers reflected on the design of Labscape, a ubiquitous computing application designed to support biologists, they found that the automatic recording and review of history was the most useful capability [19]. Biology is an experimental science, and experimental tracking before Labscape was a painstaking and error-prone manual process.

Because capturing and interacting with history is important for a number of tasks, researchers have looked at providing tracking support in databases and operating systems. The database community has focused on capturing the set of transformations that led to the current records in the database [34]. This historical record is often called data lineage or data provenance. Provenance-preserving databases allow users to ask important questions about the origin of a record in the database, such as *where* the data came from, *why* an output value resulted, and *how* the output was computed.

In the systems community, researchers have looked at tracking data provenance at the operating system level [51, 90, 97, 99]. These systems modify the operating system to link low-level system calls to changed files. Provenance information is tracked automatically and stored as additional metadata for files. Because provenance is built into the file system, programmers can leverage historical information when developing new applications. For example, Burrito is a system that uses file system provenance to automatically create an experimentation notebook for programmers interacting with data [53].

Tracking histories is especially useful for computational tasks that have a structured workflow (e.g., visualization, gene alignment). A number different of workflow tools have been created for these tasks [50]. Explicit structure can enable useful functionality that augments and enhances experimentation and exploration. For example, VisTrails, a data flow tool for creating visualizations, captures a structured data flow and associated visualizations [120]. The structure of the flow allows the tool to provide new interactions for exploring the history. For example, programmers can specify partial data flows, and the system will search the history for prior sessions that match the partial data flow.

My tools combine aspects from many different history capture and exploration tools. Hindsight tracks versions of code, data, *and* experimental results. This tool uses the structure of the data flow to help programmers navigate historical results and use those historical results to design new experiments. Hindsight automatically captures program state every time a programmer trains a new model. Because trained models have objective performance measures, Hindsight can associate each new version with a set of performance numbers. This association provides programmers with a clear overview of their progress while training a model. Programmers can also dig into these historical performance measurements in order to compare different versions.

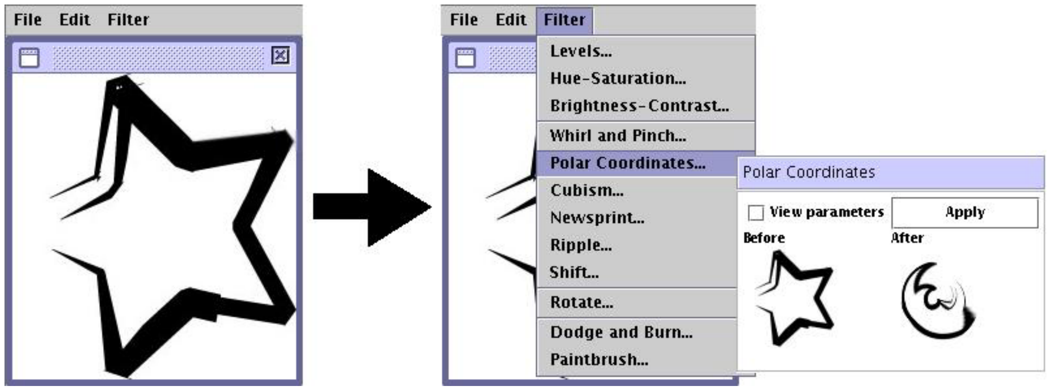


Figure 2.5: The above figure show a Side Views preview for augmenting an image. The user can see the image converted to polar coordinates side-by-side with the current image.

2.3.2 Comparing Multiple Alternatives

Comparison is a key part of all experimental processes. Terry et al. note that when working on a new design, digital artists experiment by creating variations to look at alternatives, comparing alternatives to find differences, and evaluating the goodness of the alternatives based on those differences [130]. Terry's observations are consistent with both observations of other design tasks [103] and with general guidelines for building tools that support experimentation [52, 124] and results from other studies of other design tasks. Good experimental practice affects the quality of the final design. Studies show that creating and comparing alternatives can ultimately prevent local minima and can lead to better designs in the long run [41, 42].

Tools can include support for varying a design and comparing alternatives. Terry et al. present Side Views, an interface mechanism for presenting alternatives [131]. Side Views augments current interfaces by showing the result of an action before a user takes that action. Figure 2.5 shows an example of a Side Views preview being applied to an image before the user clicks on that image. This allows the user to quickly explore alternatives without committing to the change and then undoing.

Programmers often need to compare the source code when authoring functions. Consider the scenario where a programmer finds a bug. In this situation, they may need to compare the source code for the version that has the bug to the source code for a version that is bug-free. Differences in the source code allow programmers to see what has changed [125]. Current version control tools support visualizing this difference by providing a plain text difference of the source code [64, 65]. Some tools provide graphical interfaces that show which the files that have changed and allows the programmer to look at differences between versions [133]. These differences can be helpful, but helpful differences are often drowned out by many small, meaningless differences. For example, if a programmer changes a variable name in a source code file, the file would be syntactically different but semantically the same.

Recent work has looked at providing rich comparison support for specific programming tasks. Juxtapose is one such tool that is designed to support parallel development of interfaces [55]. Programmers can create multiple alternative interfaces within Juxtapose. Similar to Side Views, Juxtapose provides side-by-side views that allow for quick, visual comparisons between alternatives. Juxtapose provides support for evolving multiple different code variations at the same time, and it can link to hardware so programmers can manipulate variations using tangible controls such as switches and sliders. Programmers can also interact with one of the interfaces to change of the internal state of the system, and Juxtapose will change the other interfaces to maintain comparisons.

I take a similar approach in the development of my tools. In Hindsight, programmers can write code, create alternatives and compare them. However, unlike the creative tasks enabled by Juxtapose, trained models provide objective, numerical metrics for correctness (e.g., accuracy, precision, and recall). These metrics can be used to quantify the differences between alternatives. Programmers can make deeper comparisons by looking at changes in the predicted labels for individual examples.

2.4 Current Machine Learning Tools

Given the success of machine learning as a tool for solving hard problems like document classification and medical diagnosis, research and industry are trying to make it easier to training new models with machine learning. Current approaches can be broadly categorized into two groups: domain-specific tools and general-purpose tools. Domain-specific tools reduce the complexity of the machine learning process by limiting the types of input a user can provide and focusing on support for a single domain (e.g., image classification or text recognition). General-purpose tools support specific tasks (e.g., generating features, choosing an algorithm), but do not effectively support the process.

2.4.1 Domain-specific Tools

There is a growing body of work on tools that help people train models for specific domains. In this dissertation, I refer to these systems as *domain-specific tools*. Crayons is an early example of a domain-specific tool [45, 46]. This tool allows application designers to train image segmentation models, which are used to create camera-based interfaces. Crayons does this by allowing designers to directly interact with an image using a "painting" metaphor. In this section, I describe how a potential user would use Crayons to train a model in order to show how domain-specific systems work. I then provide examples of how domain-specific tools have evolved since Crayons and compare domain-specific tools with my tools.

How domain-specific tools work

Figure 2.6 shows how designers interact with Crayons to segment an image. In this scenario, the designer is training a hand tracking system. To track a hand, the system first needs to recognize which pixels in the image correspond to a user's hand and which pixels do not. In Crayons, designers separate the pixels that correspond to the hand from the pixels that correspond to the background. In order to make this separation concrete, the designer selects an image from the image stream and then colors over the pixels that correspond to the hand. The designer then selects another color, colors the background, and asks Crayons to learn a model that segments the hand from the background. Coloring provides Crayons with labeled pixel data, and Crayons learns an image segmentation model based on the colors. Then, Crayons segments the image using the new model and overlays the results on top of the original image. The designer can see the results, find areas where the image segmentation algorithm is not working well, and add more data by recoloring the image. They continue this process until the image segmentation model effectively segments the image.

The design decisions behind Crayons are important for lowering the barrier to using machine learning. Crayons focuses on a specific domain: image segmentation for camera-based interfaces. Limiting the application domain makes building Crayons possible. A limited domain allows Fails and Olsen to provide targeted visualizations and interactions. In Crayons, the coloring interaction for gathering pixel data and the overlays for checking

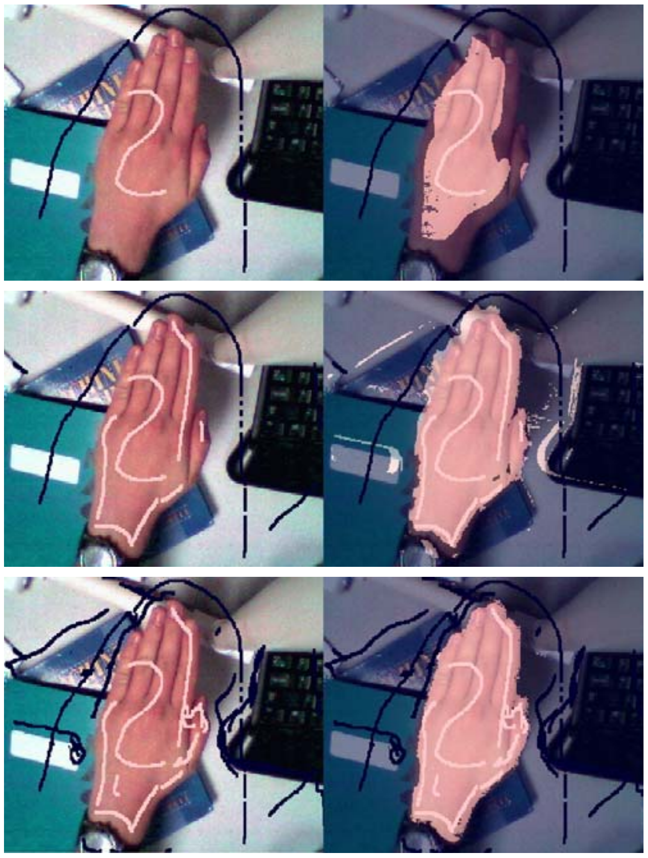


Figure 2.6: This figure presents stages of training an image segmentation model using Crayons. The designer iterates between providing the learning algorithm with data by coloring the image (right) and checking the effectiveness of the model by looking at the overlay of the pixels (left). Over time they are able to develop a more accurate model.

segmentation results are particular to the problem domain of image segmentation.

By focusing on the image segmentation domain, Crayons can hard-code feature generation and classification algorithms that work well for segmenting images. Removing complex steps in the machine learning process makes Crayons accessible to more people. Because the designer does not need to write code to create visualizations, generate features, and train a model, non-programmers can effectively train image segmentation models.

Crayons provides an interactive loop where the user can quickly provide more data for the learning algorithm, and Crayons can quickly provide feedback on what it understands. For domain-specific tools, the feedback does not need to be rapid [63], but in many cases it is. What is important is that there is a feedback loop. The user needs to communicate what they want the system to do. In Crayons, users provide examples by coloring images. The computer needs to communicate what it knows to the user. In Crayons, computers provide overlays on top of images. Based on this feedback, the user adjusts the information they provide and thereby guide the system to a useful model.

Further work on domain-specific tools

Researchers have taken the interactive machine learning model and applied it to their own domains. For example, Exemplar allows designers to build sensor-based interactions [54]. Designers demonstrate interactions using a physical device connected to a computer. Exemplar visualizes the sensor stream, and designers can annotate segments of the visualization that correspond to the interactions. Using these annotations, Exemplar learns a model that it applies in real time to the sensor stream. Designers can immediately test the model by demonstrating the interaction again and checking the visualization to see if the interaction was properly labeled. Exemplar is one of many successes that have come from applying the interactive machine learning model to a specific problem domain. Interactive machine learning has also been applied successfully to a number of other domains such as computer vision [91], text extraction [62, 111, 135], instrument creation [47], network alarm triage [17], document clustering [21, 43], context-aware applications [40], 3D gesture recognition [20], and image search [49].

Much of the work on domain-specific tools takes the form of an existence proof. This work shows that it is possible to build tools that enable people to train machine learning algorithms without having to write code. However, this work does not show how these kinds of systems should be built in general. Recent work has focused on the general design principles behind interactive machine learning systems. Stumpf et al. discuss what types of explanations are the most understandable [127]. Amershi et al. show how to pick examples that do the best job of communicating to the user what the computer understands [14], how to help users keep track of models as they experiment [15], and how to provide feedback on which features the system is using [16].

Designers of domain-specific tools trade generalizability for reduced complexity. By reducing complexity, domain-specific tools increase the number of people that can actually use machine learning to solve problems. However, the same design decisions that reduce the complexity tend to make domain-specific tools brittle. For example, by hiding certain tasks in the machine learning process (e.g., data processing, feature generation, and algorithm selection), creators of domain-specific tools limit the space of problems that can be solved with those tools. As data collection technology improves, there will always be new domains that could benefit from machine learning techniques. Tools such as Crayons cannot help in those domains.

Hardcoded algorithms and visualizations can even prevent domain-specific tools from handling simple variations of problems within their own domain. Consider the situation where a Crayons programmer finds that lighting plays a huge role in the accuracy of their model. Crayons itself is not robust to lighting changes, but a new feature could easily make it more robust. But, the design of Crayons prevents the programmer from changing any of the features – they're stuck. They must start from scratch and rebuild the same framework provided by Crayons in a general-purpose programming tool.

By focusing on support for general-purpose programming, I provide a flexible, domain-agnostic environment in which programmers can represent a wide variety of different problems. For instance, both Hindsight and Gestalt allow programmers to write code that defines every step in the classification pipeline. My studies show that every step in the machine learning pipeline is important when trying to build an model for a new problem. Programmers need to be able to change their dataset, their feature generation code, their machine learning algorithms, and their evaluation criteria.

General-purpose tools will not subsume the need for domain-specific tools. There will always be the need for domain-specific tools in important domains where the programming skill is low and domain impact is high. Nevertheless, general-purpose tools can provide design guidelines and new feedback mechanisms for domain-specific tools. For instance, Prospect provides a new way for programmers to understand their dataset. Understanding a dataset is also important for non-programmers using domain-specific tools. A domain-specific explanation layer on top of Prospect's visualizations can provide non-programmers with valuable feedback on how to improve their model by providing better data.

2.4.2 General-purpose Machine Learning Tools

General-purpose support for machine learning can be separated into three categories. First, there are a set of tools that grew out of programming language and programming environment support. Second, there are tools that grew out of API support for machine learning algorithms. Finally, there are tools that grew out of the interactive machine learning and interactive visualization work. All of these approaches are general, but none of them provide support for the process of training a model. In this section, I discuss these tools and contrast them with my work.

Programming Languages and Programming Environments

Researchers have looked at how to support machine learning with new programming languages. These languages have special constructs that allow programmers to easily represent common concepts or perform common actions. Machine learning specific constructs allow the system to optimize the performance of commonly used functions and reduce the amount of boilerplate code that programmers have to write.

Learning Based Java is one such language [112]. It reduces boilerplate coding by creating language constructs that support common tasks, such as exploring a range of parameters. This language allows programmers to iterate quickly, reducing the computational cost of training a new model through automatic caching of intermediate results.

The distinction between my tools (i.e., Gestalt, Hindsight) and programming language support for machine learning, is the same as the distinction between traditional programming languages (e.g., C#, Java) and development environments (e.g., Visual Studio, Eclipse). Tools like integrated debuggers and static analysis techniques are independent of language, and help support the programming process. Similarly, Gestalt's support for visual analytics and Hindsight's support for history tracking are independent of the underlying language and provide complementary support to programming language support for machine learning.

Gestalt and Hindsight are similar to numerical computing environments such as Matlab [5] and R [66]. These tools provide both programming language and development environment support for math and statistics. These tools are used extensively by machine learning community because they make it easy to write code to implement a machine learning data flow and create visualizations to analyze data. These tools have inspired packages for existing traditional programming languages that provide similar functionality [3, 6].

Since coding and data analysis happen in the same place, programmers using these numerical computing environments do not have to switch between applications, nor do they have to write glue code that converts data between formats. Machine learning programming is easier with numerical environments than with most other programming environments. However, these tools are not designed to support machine learning specifically. For example, because it is made for math, Matlab's visualizations mostly focus on charts and plots of matrices and vectors. With machine learning tasks, the appropriate visualization depends on the dataset, and these matrix-oriented charts and plots do not apply to many common datasets. Therefore, numerical computing environments make it difficult to build new visualizations of a dataset.

Visualizations in numerical computing environments are also not interactive. To move between visualizations or to dig down into visualizations, programmers have to write additional scripts. The difference between my tools and numerical computing environments is similar to the difference between command line debuggers and breakpoint debuggers. Command line debuggers can help experts better understand the behavior of the system, but they are cumbersome and hard to use. Making debugging easier with visual debuggers encourages more programmers to debug and makes the debugging process smoother and easier. By providing interactive visualizations, my tools help programmers focus more on data and less on code.

Finally, when training a model, programmers organize their code in a common data flow structure. Numerical computing environments provide some data flow support (e.g., Simulink in Matlab), but they don't take advantage of that structure to support programmers training a model. For example, Matlab doesn't use this structure to store intermediate results like Gestalt or track experimental history like Hindsight. My work shows that supporting and leveraging the machine learning data flow can help programmers better organize code, track experimentation, and localize errors.

API support

Because implementing a bug-free machine learning algorithm from scratch is hard, there are many different APIs that provide libraries of commonly-used algorithms [38, 137]. These libraries remove an important barrier to using machine learning. If a machine learning algorithm is poorly implemented, it may still provide a workable model, albeit one that performs poorly. Attributing poor performance to an incorrectly implemented machine learning algorithm is difficult, because understanding how machine learning algorithms work requires in-depth knowledge of the statistics and math that power the algorithm. By removing the algorithm as a potential source of error, these APIs reduce the debugging burden and increase the population of programmers that can use machine learning algorithms.

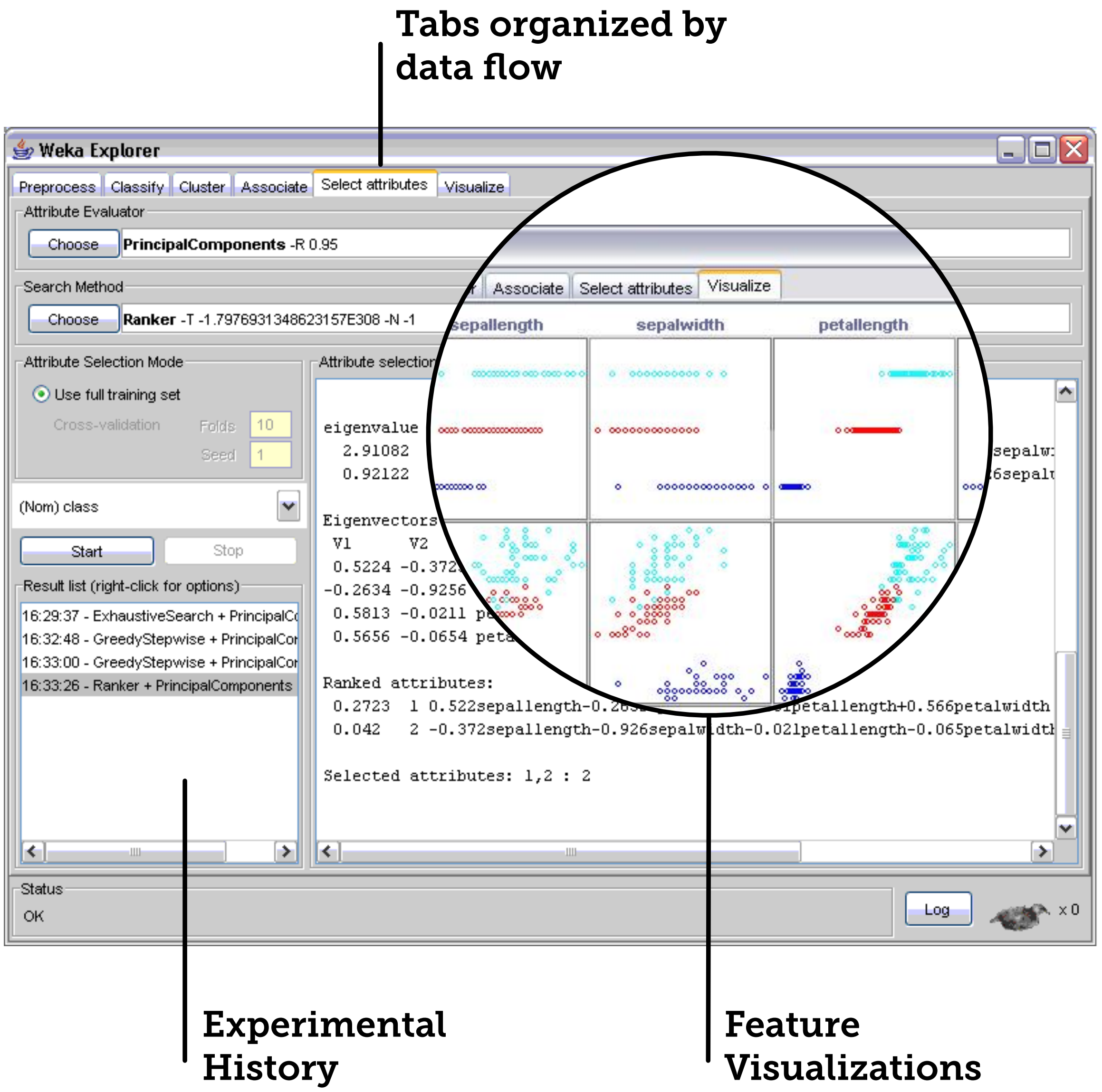
Bug-free algorithms are only one of many different factors that determine the behavior of a trained model. Effectively training a model also involves working with data. As mentioned previously, errors in data are common. These errors affect the behavior of the system. Programmers need a way to understand relationships between code and data. They also need to be able to experiment to improve the performance of their program. The designers of some APIs have recognized this problem, and have developed interfaces on top of their APIs to provide some visualization and experimentation support.

For example, Weka provides an experimenter interface as shown in Figure 2.7. This interface allows programmers to load datasets, filter features, and choose an algorithm. It keeps track of the different algorithms that the programmer has tried and their corresponding results. Weka also supports common machine learning visualizations such as ROC curves and scatter plots of feature responsiveness.

Weka's experimenter interface is restricted to a set of static steps. Tools such as Knime and Rapid-Miner extend Weka by providing an interface for specifying arbitrary data flows [24, 93]. Programmers build a pipeline by connecting previously implemented components that load data, process data to generate features, train models, and visualize results.

However, these tools have the same limitations as other data flow tools. For example, there are a number of domains where data flow tools will not be able to provide working components. To build more components, programmers must leave the tool, start up a standard development environment, write code, compile the component, and add it to the tool. This process happens every time the programmer needs to create a new component or iterate on a component they have created. A general-purpose tool that supports training a model must support writing code as well.

Additionally, these tools often require programmers to convert their data into a common format. Programmers have to write the data-processing code somewhere else, but it is hard to connect the raw data back to the model after converting it into a common format. General-purpose tools need to support the inspection of raw data in order to help programmers detect errors in their data and come up with new features.



Attribute Evaluator: **PrincipalComponents** -R 0.95

Search Method: **Ranker** -T -1.7976931348623157E308 -N -1

Attribute Selection Mode: Use full training set; Cross-validation Folds 10 Seed 1

(Nom) class

Result list (right-click for options)

```
16:29:37 - ExhaustiveSearch + PrincipalC
16:32:48 - GreedyStepwise + PrincipalCor
16:33:00 - GreedyStepwise + PrincipalCor
16:33:26 - Ranker + PrincipalComponents
```

Attribute selection output

```
eigenvalue
  2.91082
  0.92122

Eigenvectors
 V1       V2
 0.5224 -0.372
-0.2634 -0.9256
 0.5813 -0.0211 pe
 0.5656 -0.0654 peta

Ranked attributes:
 0.2723   1 0.522sepallength-0.26                  petallength+0.566petalwidth
 0.042    2 -0.372sepallength-0.926sepalwidth-0.021petallength-0.065petalwidth

Selected attributes: 1,2 : 2
```

Status: OK

Figure 2.7: The Weka experimenter interface provides programmers with some support for visualizing data and tracking experiments. The tabs are organized according to distinct tasks such as choosing a model and selecting features. Users can look at the history list to the bottom left to see which models they have trained.

Other APIs provide support for feature generation and feature selection. Although focused on the domain of sensor based activity recognition, the SUBTLE toolkit provides a generalizable technique for automated feature generation [48]. Typically, programmers generate a number of different feature combinations and then filter that number to pick only the most discriminative features. SUBTLE automatically generates features using simple operators on base features and automatically filters features using wrapper-based feature selection. Although useful, programmers still need to write code to create basic features, and the new features are limited to combinations created by applying operations provided by the API.

Interactive Visualizations of Results

When training a model, there is a level of indirection between actions and results. For example, a programmer training a digit recognition system may observe that the system is unable to distinguish between fours and nines. They may change parameters to their classification algorithm and observe the results to see if the performance improves. They may prefer to make errors on other digits in order to improve the performance of fours versus nines. Traditionally, there hasn't been a way for programmers to directly specify these preferences.

Ideally, this indirection would be removed. The programmer could specify that they want better performance on specific classes, and the model would change based on those preferences. There is recent work that provides avenues for programmers to interact directly with the output of a model in order to improve performance. This work focuses on creating new algorithms and interactions that work together to remove indirection. The programmer indicates through some interaction what they want the results to look like and the algorithm modifies itself based on those interactions [70, 129].

Talbot et al.'s EnsembleMatrix is one such tool [129]. An ensemble is a meta-classifier that combines results from many different models. EnsembleMatrix trains an ensemble for large multi-class problems. It then provides an interactive confusion matrix view that can be used to steer the accuracy of the classifier. The programmer steers the classifier by grouping regions of the confusion matrix.

ManiMatrix builds on this idea by having programmers interact directly with a confusion matrix visualization to help guide the model [70]. Programmers can indicate preferences by clicking on cells in the matrix. These preferences are translated into a cost matrix. The underlying machine learning algorithm then optimizes this cost matrix in order to automatically guide the development of the model.

These tools take a complementary approach to my tools. The visualizations and interactions provided by these tools can be integrated into Gestalt and Hindsight. However, programmers will still need support for writing code, support for collecting data to build their machine learning program, and support for analyzing code in order to understand the behavior of that program.

2.5 Summary

The machine learning process is characterized by three tasks: authoring code, working with data, and running experiments. General-purpose solutions exist for all three of these tasks, and domain-specific solutions support the machine-learning process for targeted data domains. However, there is little general-purpose tool support for machine learning.

Before new tools can be built, it is important to observe and understand both the machine learning process and how programmers use current toolsets to train models. This understanding can guide the development of new tools designed to support the general-purpose application of machine learning algorithms. In the next chapter, I present two studies that examine the difficulties that programmers face when using machine learning. My subsequent chapters describe tools that I have built based on the results of my studies.

Chapter 3 | **STUDIES**

Programmers know how to author functions by writing code. On the other hand, training models by collecting data, writing code, and analyzing results is relatively unfamiliar. In this chapter, I study the difficulties programmers face when tasked with the unfamiliar process of training models by using machine learning algorithms.

For the purposes of this dissertation, I define **experts** as programmers with a machine learning background and experience training models using machine learning algorithms. Experts know how machine learning algorithms work and have experience structuring a machine learning data flow, collecting data, running experiments, and understanding the relationships between data and results. I define **non-experts** as programmers that have exposure to learning algorithms through an introductory course or through self study but are not skilled at applying them.

My studies were motivated by the following questions:

- What kind of problems are programmers trying to solve with machine learning?
- What process do they follow to solve these problems?
- What tools do they use to help them with this process?
- How is the process for training a model different than the process for authoring a function?
- What are the barriers that both non-experts and experts run into when training a model?
- How do experts overcome the barriers faced by non-experts?

- How might current programming tools better support training models?

To address these questions, I interviewed computer science researchers, both experts and non-experts, about training models using machine learning and creating applications that use those models. These interviews informed a laboratory study in which participants used a typical machine learning toolchain to train a model. These studies reveal breakdowns in the programming process and limitations of current machine learning tools. In this chapter, I discuss my methodology for both studies, results from these studies, limitations of the studies, and implications for the design of new tools that support machine learning.

3.1 Interviews

I started my exploration of the use of machine learning by interviewing programmers who had successfully built applications based on machine learning models. In this section I discuss my interview methodology and provide results.

3.1.1 Participants

I interviewed 11 researchers with experience using machine learning to train models. I was looking for researchers who were trying to train models with the end goal of using those models in applications. These researchers are in contrast to machine learning researchers who are trying to create new algorithms or study the performance of existing algorithms. With this goal in mind, I interviewed both experts and non-experts. The non-experts in my study were human-computer interaction researchers who were looking for machine learning solutions to their problems. The experts were machine learning researchers who were looking for problem domains in which to apply their algorithms. In this chapter, I refer to my interview participants as IP1 to IP11.

These researchers have worked on a wide variety of problems, spanning a range of data domains and machine learning techniques. As an indication of the breadth of their experience, I note that these researchers have worked on problems such as: intelligent digital photo management, vision-based facial expression recognition, availability modeling in instant messaging, EEG-based recognition of brain activity, RFID-based activity recognition for elder care applications, accelerometer-based activity recognition for fitness applications, mixed-initiative pen-based text input, programming-by-demonstration approaches to text editing, interactive tools for creating camera-based interfaces, automated network packet diagnosis, and models of musical style. Each interview participant has published multiple papers in top venues.

3.1.2 Procedure

Each participant recalled two to three prior projects in which they trained models using machine learning. I asked them to pick one project and describe the lifecycle of building the project, from conception to completion. I also asked them to recall how the use of machine learning applied to other aspects of their project, such as the design of the application and collaboration with other programmers.

Participants were provided with a notepad and pen and asked to diagram their process while they discussed the process. I encouraged participants to diagram in two ways. First, in the event that they stopped diagramming, I reminded them. Second, I engaged in the diagramming process myself. I provided by own annotations and edits to their diagram to help clarify my understanding of their process. After discussing the first project, participants compared and contrasted it with other projects they had initially discussed. Interviews lasted for 40 to 90 minutes. I recorded the audio from each interview and transcribed the recording for further analysis.

3.2 Intermediate Interview Results

My interview participants described structuring their machine learning code into a linear data flow. I call this data flow a pipeline. The pipeline includes steps for loading data, processing data, training a model, and evaluating that model. Figure 3.1 shows a two examples of pipelines that interview participants sketched when describing their process.

My participants used a variety of common machine learning techniques such as classification, regression, and clustering. Because the pipeline for those techniques is similar and because classification was the most popular of those techniques, in this section I focus on the classification pipeline. This classification pipeline is created and refined through a process in which programmers run experiments to explore the space of possible models. In this section, I describe both the structure of the classification pipeline and the process. I leave a full analysis of my interview results for the combined results section (section 3.4) later in the chapter.

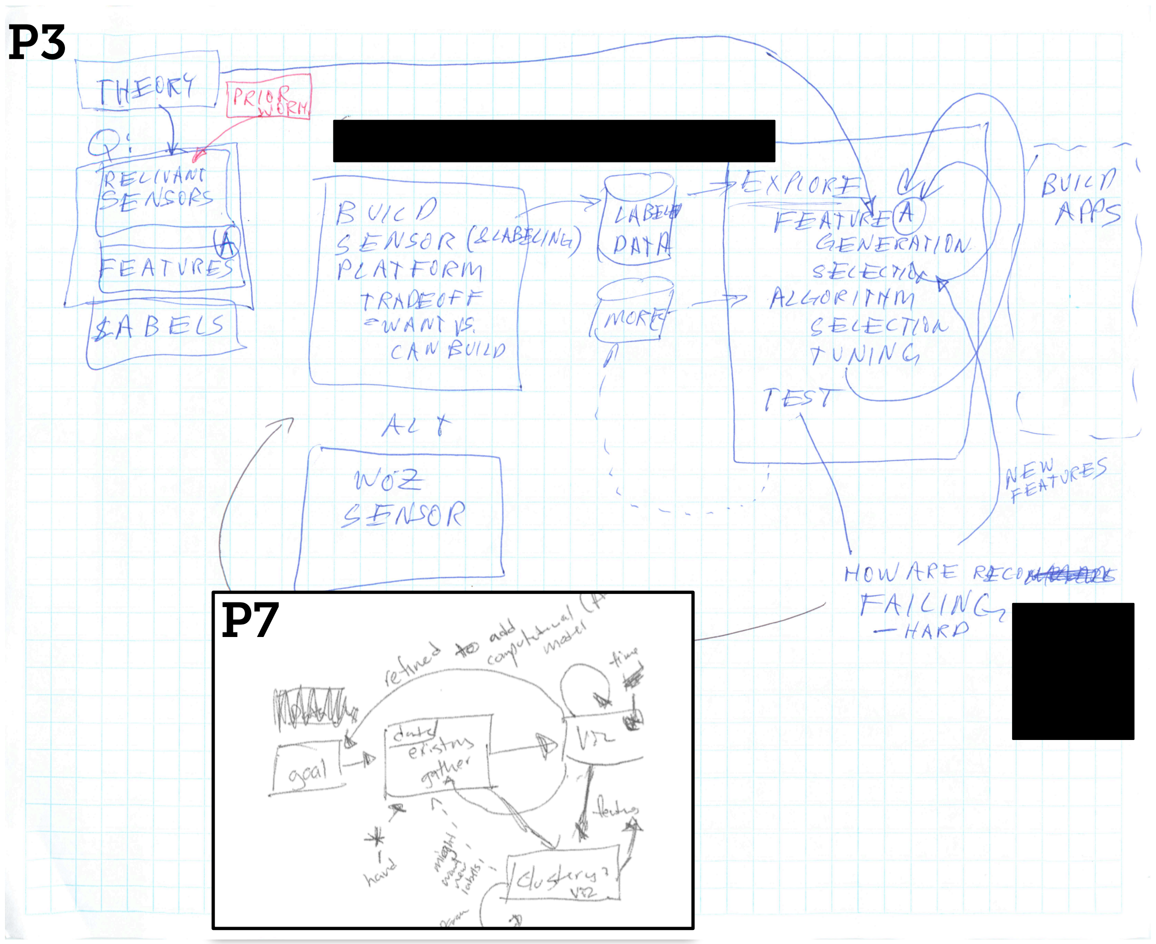


Figure 3.1: The figure above shows sketches from two interview participants. P3 was working on a classification problem and P7 was working on a clustering problem. Note that the high level steps of problem formulation are the same (P3 calls it theory and P7 calls it goal). Both involve collecting data (sensors for P3 and data for P7). And both have an iterative process for improving the performance of their model.

3.2.1 The Classification Pipeline

The **classification pipeline** is a linear sequence of steps that starts with formulating a problem. Programmers must define the inputs and outputs of their function. After they define inputs and outputs, they can collect data. Because classification is a supervised technique, programmers also need to provide a ground truth label for each example in their dataset. Labeling can be a difficult hurdle to overcome because it is both time consuming and error prone.

Consider the scenario where a programmer is training a model that classifies news stories. This model would take as input a news story from a website (e.g., NyTimes, Engadget) and output the category of story (e.g., finance, public interest, politics). Internet portals, such as Google and Yahoo, use similar models to aggregate stories from the internet. To build such a classifier, the programmer must have someone read each story and group stories into categories. Some categories may be too similar for people to effectively distinguish (e.g., politics and finance overlap on regulation articles). This can lead to erroneous labels. Labeling is also expensive, because a person has to spend time reading and grouping articles.

Figure 3.2 shows the steps in the classification pipeline after the programmer has labeled data. Given a labeled dataset, the programmer must *parse* the raw data. Parsing involves loading, processing, and cleaning the data. Processing and cleaning involves steps such as splitting data, combining different data sources, and sampling the dataset to get a subset with an even distribution of classes. For web news classification, the programmer must combine data from different news websites. Because websites have different formats, a programmer must create site-specific loaders to load articles and convert them into a common format.

After they have a parsed dataset, programmers can now *generate features*. Features tell the machine learning algorithm what facets of the data are important. Programmers are often experts in their data domain, which means they can describe features that may distinguish classes in their dataset. Programmers need to encode that knowledge into a numeric or categorical value that the machine learning algorithm can use to model the data. Encoding this knowledge often involves writing code. For web news classification, a programmer may find that proper nouns do not overlap between various news categories. The people and places mentioned in sports articles are not the same as those mentioned in politics articles. They can turn that understanding into a generally discriminative feature (e.g., counting the number of times each proper noun appears).

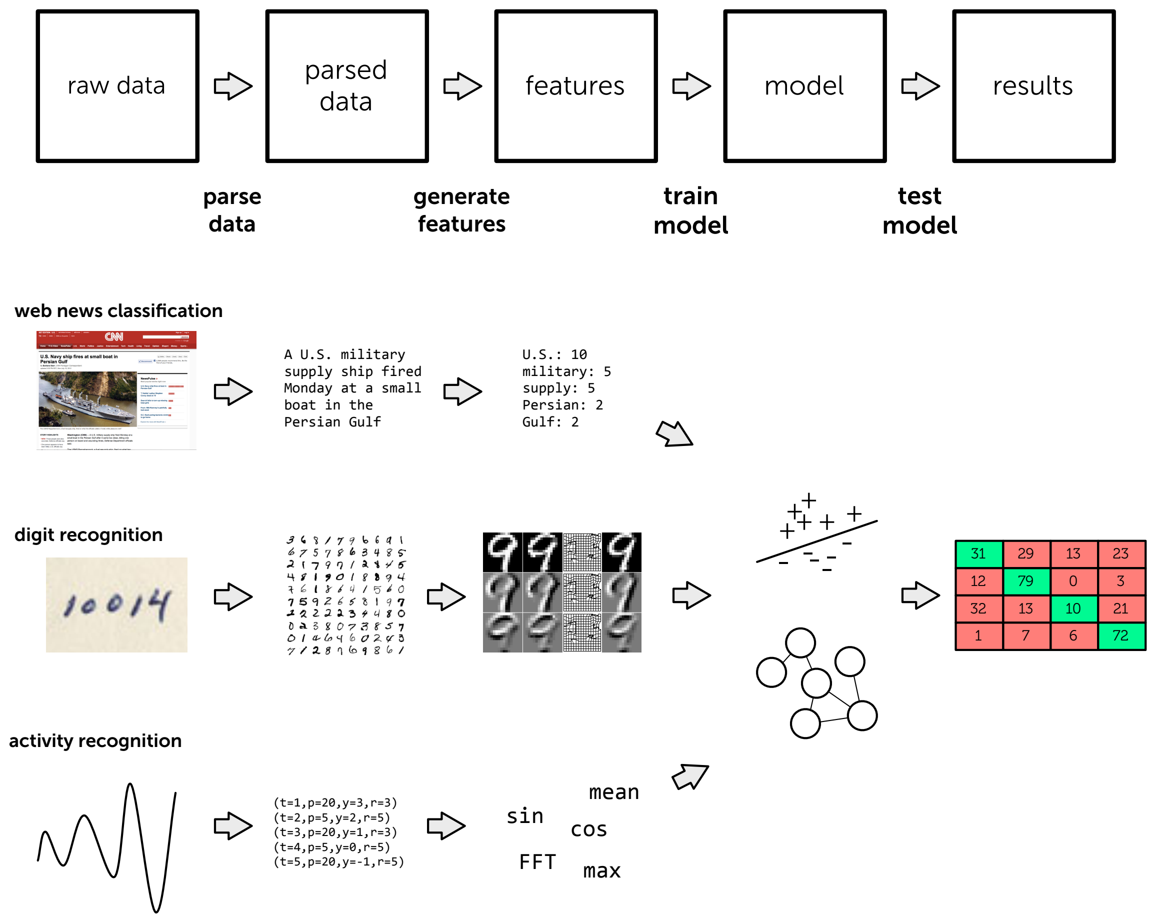


Figure 3.2: Web news classification, handwritten digit recognition, and sensor based activity recognition have similar classification pipelines. Programmers must parse data and generate features to get the examples into a form a classification algorithm can understand. After the data is in the right form they can explore different algorithms and run experiments.

After features are generated, programmers can finally pick an algorithm to *train* a model and design an experiment to *test* the model. A common experiment is random cross-validation, where the data is randomly spilt into n segments or folds. All but one of the segments (i.e., $n - 1$ segments) are used to train a model, and the last segment is used to test the model. The process is repeated until all n segments are used as the test set. The end result of this process is a model and measurements that describe the model's performance.

3.2.2 Machine Learning Process

The process of creating a classification pipeline is exploratory. Programmers train a model by gathering data and writing code. They test the model by running experiments and analyzing data. Experiments such as cross-validation provide high-level measurements such as accuracy. Programmers understand the behavior of the model by combining these measurements with other information, such as the raw data, the feature values, and the parameters to a learning algorithm. Based on their understanding, programmers change their pipeline and run new experiments. They continue searching through the space of possible models to find a model that works for their problem. I refer to the exploratory process as the *machine learning process*.

3.3 Laboratory Study

I designed the digits task to examine how difficulties manifest as programmers work on a problem. The task is based on descriptions of the process provided by my interview participants. This section presents the participants, task, development environment used by participants, and experimental procedure of my study.

3.3.1 Participants

I recruited ten participants, all non-experts. To avoid confusion when discussing the two groups of participants, I refer to laboratory study participants as LP1 through LP10. After early experience showed that people with no experience using machine learning could make very little progress, I decided to pull from a population that had some exposure to machine learning. All of the participants in the study were graduate students who had some classroom experience using machine learning.

They were also familiar with the toolset provided in the study. All of the participants reported prior experience programming in Java and using the Eclipse IDE. Four out of the ten had previously used Weka, and three of those four had worked with Weka's API. Because I was interested in experienced programmers rather than novice programmers, this population is consistent with the goals of my study.

3.3.2 The Digits Task

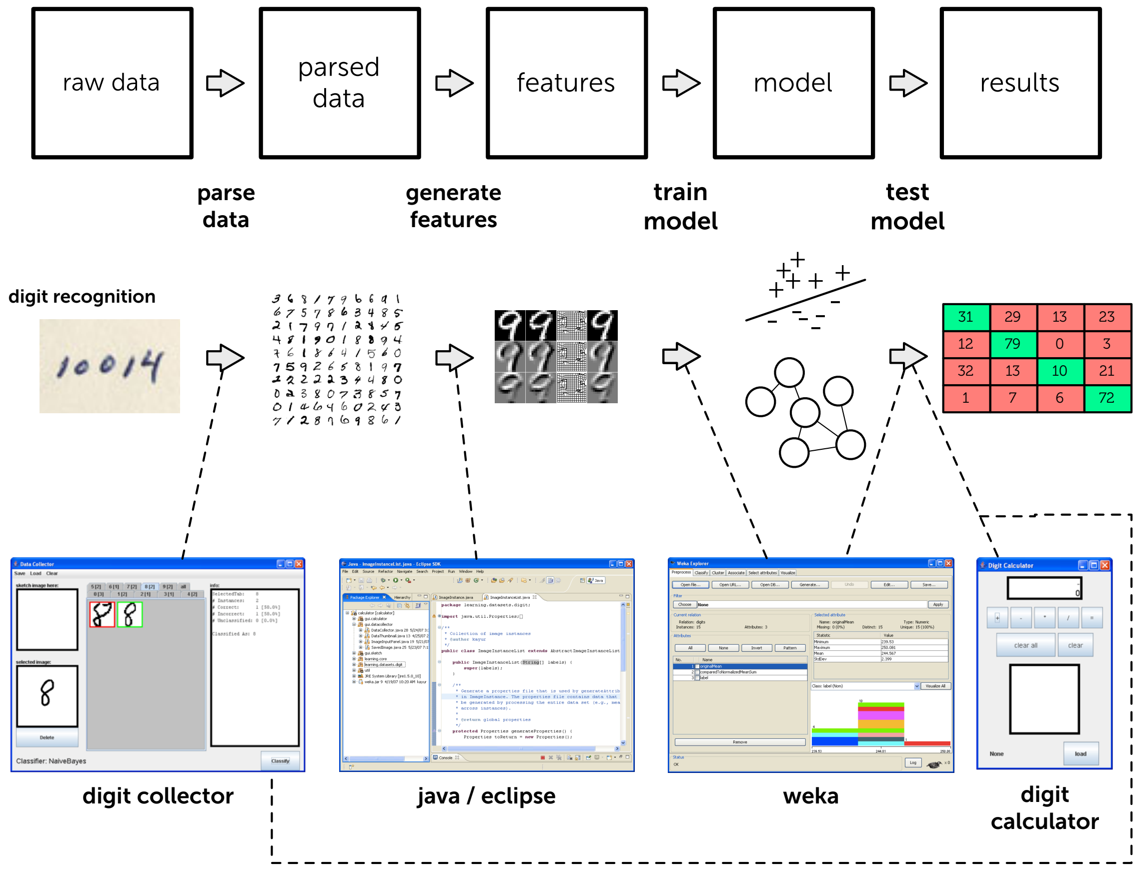


Figure 3.3: The figure above shows the handwritten digit recognition classification pipeline and the toolset for the digits task. I provided participants with four tools used for different tasks. The dashed lines show what the tools were used for. Participants collected and processed data in the digit collector. Participants generated features by writing Java code in Eclipse and selected an algorithm and trained a model using the Weka Explorer interface. Participants tested their model in three ways: looking at the output in Weka, interacting with a simple digit calculator application, and loading their model and testing data in the digit collector.

Participants in my study were given the task of building a handwritten digit recognizer approximately four hours. An overview of the task can be found in Figure 3.3. I chose this task for many reasons.

First, handwritten digit data is easily interpretable by most people and easy to collect in a laboratory study. Unlike other classification tasks, such as activity recognition or medical image analysis, I do not need my participants to be experts in a data domain nor did I need them to leave the study environment to collect data. And because images have a natural visual representation, the accuracy of the classification is easy to verify. In contrast, text data may be easier to understand, but takes much longer to verify. This natural interpretability of digit data was critical to making the classification problem tractable in the four hour time span.

Second, many people have worked on handwritten digit recognition over the years, and as such there was as wealth of information that participants could leverage by going to the internet and searching for related work. Also because there have been many solutions over the years, participants could reach a satisfactory solution through a number of different paths. Participants could collect more data, build better features, and refine their algorithm. All of those actions could lead to better results.

Third, a good experimental setup for the digit problem is hard to do in current tools. For example, most tools assume that random cross-validation is a valid way to evaluate a model. However, my interviews show that for certain data random cross-validation can provide good evaluation results, but the trained model perform poorly on new data. Digit recognition is an example of a dataset. I wanted to see if participants could recognize the limitations of default settings and change their experimental setup. In the task, participants were provided with more data halfway through the study and an interactive calculator application to help them test with their model with fresh data.

Evaluations are based on more than just accuracy. In my interviews, I found that programmers cared about factors such as interactivity. The digit task lends itself to an easy, interactive application that can be evaluated in a laboratory study. The interactive calculator application also allowed me to test interactivity in addition to accuracy. I discuss this finding in more detail in my results.

3.3.3 The Development Environment

Participants worked using a toolset that included the entire process of collecting training data, extracting features from the images, choosing a classification algorithm, and then interactively testing their model in a simple application. This toolset is illustrated in the bottom part of Figure 3.3. This toolset is typical of the tools that interview participants discussed using in their work. It includes four components.

The first component is a data collection application. The design of this application is similar to that of existing pen-based gesture recognition systems [87]. Participants inputted digit data using a stylus and a 1024x768 Wacom Cyntiq tablet display. By selecting the tab corresponding to a digit between 0 and 9, participants could create labeled instances of that digit. Digits were captured as 125x125 monochrome images and as normalized 20x20 grayscale images (see Figure 3.3). Participants could create and manipulate multiple data files containing such images.

The second component is the Eclipse development environment for Java, used to extract features from the digit images. Existing tools generally parse files into simple tabular formats. Interview participants reported using these tools because they reduce the effort needed to program and debug a machine learning algorithm. Interview participants also used simple programs that they glued together to parse data, compute features, and store those features in a tabular format. To avoid the debug and development cost of these small components, I provided data conversion code. Specifically, I provided participants with a codebase that parsed the file format used by the data collection tool, provided stub functions for compute features based on the digit images, and code to output the features in a format the next tool could understand. As a part of introducing the study, participants were shown where and how to edit the feature generation stubs.

The third component is Weka, a well-known and widely-used machine learning tool [137]. Weka provides a large library of feature visualizations, filters, and learning algorithms. Participants used Weka to load the tabular feature files output by their feature generation code, to apply learning algorithms, and to evaluate model accuracy using standard evaluation techniques (such as random fold cross-validations). Weka is the one of three places where participants can evaluate their model.

The last component is a simple interactive calculator application used for testing the system. This application used the participant's feature generation code and a Weka-exported model to interactively classify pen input. The calculator application is the second place where participants can evaluate their model, and I included it for two reasons. First, many interview participants reported that "trying it out" was an important part of their process of applying machine learning. Providing this application closed the loop and allowed participants to interact with the model they created. Second, the interactive use of a model exposes important aspects of that model that are not captured when only considering accuracy, most notably the computational cost of the features used by the model. Participants were told their models needed to work well in an interactive context. If the features or the algorithm used by a participant were too computationally expensive, a noticeable lag would result.

Finally, participants could also evaluate their model a third way by using the data collection interface. Given a the model and an existing dataset, the data collection interface would highlight examples that were classified correctly. Programmers could hover over the example to get the predicted value of the classification. The digit collection interface facilitated the creation of multiple datasets, training a model on one of the datasets, and testing it on another.

3.3.4 Procedure

Participants worked in a small office free of distractions, and they were asked to think aloud as they worked. Links to the Java API and the Weka API were provided, and participants were free to use any resources they felt would be helpful. Nearly all chose to use the internet to find information about features, and several downloaded code to compute features. The workstation included a 24" Dell 2407WFP display running at 1900x1280 and the Wacom Cyntiq tablet display running at 1024x768. Participants were free to use the available monitor space however they chose. Most used the primary display for interacting with the Eclipse development environment and with Weka, thus using the tablet display almost exclusively for interacting with the pen-based data collection and calculator applications. Commercial screen capture software continuously captured the desktop, a video camera recorded the physical environment, and custom software took continuous snapshots of the files in participant workspaces. To minimize the impact of the computational demands of machine learning tools and the screen capture software, participants worked alone on a computer with two quad-core Xeon processors (at 2.66 GHz) and four gigabytes of RAM.

Each session started with a tutorial, familiarizing participants with the toolset by stepping them through the collection of a handful of labeled digit images, the extraction of a simple feature, the creation of a simple model, and the interactive use of that model in the calculator application. They were then given two hours to collect data, develop features, and create the best classifier they could. After a break, I provided participants with 200 labeled

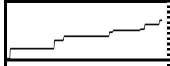
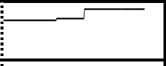
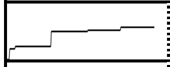
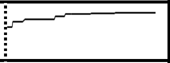
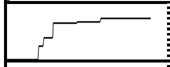
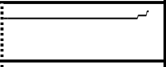
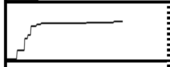
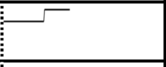
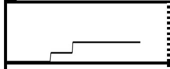
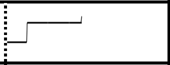
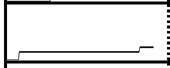
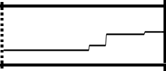
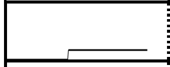
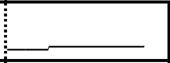
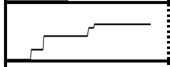
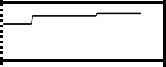
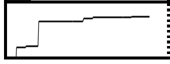
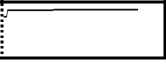
	First 2 Hours	Second 2 Hours	Final Accuracy
LP1			84.7%
LP2	----	----	75.3%
LP3			78.3%
LP4			82.9%
LP5			84.7%
LP6			78.0%
LP7			56.9%
LP8			22.8%
LP9			78.8%
LP10			84.4%

Table 3.1: The figure above shows the final accuracy of each laboratory participant's model, as well as a plot of the accuracy of each participant's model over the course of each two-hour session.

digit images collected from four different people (5 examples of each digit per person). This data was provided to see how participants would use it in developing their system. Participants then had another two hours to continue developing their system, using both the new data I provided and data they collected. Participants were told that their classifier would be evaluated according to its accuracy for digits sketched by other people, subject to the constraint that it worked fast enough for interactive use. Participants received a $50 gift certificate for participation, and the participant who created the best model received an additional $50 gift certificate.

3.3.5 Laboratory Results Overview

Table 3.1 presents the final accuracy of each laboratory participant's digit recognition system, as well as a plot of the evolution of the accuracy of each participant's system over the course of the study. These accuracies were computed using 2000 labeled digits collected from 20 different people (10 examples of each digit per person), none of which were provided to any of the participants. The plots were computed by scripts that created and tested models based on automatically captured snapshots of participant workspaces. I am unable to plot the accuracy of LP2's models over time, because LP2's system made heavy use of a set of files outside the environment that was captured.

3.4 Results

My interviews show that training a model involves constructing a pipeline through which raw data is turned into a model. Programmers follow an experimental process when building that pipeline. This section expands on those observations. It combines results from both interviews and laboratory studies to break down what programmers are doing when they use machine learning to train a model, the difficulties they face, and the breakdowns in current tools.

3.4.1 Programmers Iterate on a Pipeline

Training a model involves building a pipeline. The pipeline processes data to get it in a form that a machine learning algorithm can understand. Each step in the pipeline depends on the previous step. Programmers must formulate their problem to understand what type of data to collect. They then must collect and parse data, which can involve combining multiple data sources, labeling data, and cleaning noisy data. After they process data, they must generate features. Programmers can choose a learning algorithm and train a model only after all of these dependences have been met.

A linear pipeline doesn't mean a linear process. Programmers can't assume they are done with a step in the pipeline. Revisiting past choices is important. Programmers must collect more data, write code to generate new features, or choose a different classification algorithm. Training a good model involves revisiting and refining all of the steps in the pipeline.

Interview participants emphasized the non-linearity of the machine learning process when discussing their work. Dead-ends were overcome by revisiting an earlier point that they had assumed was working correctly. For example, IP6 described a long and fruitless exploration of learning algorithms in collaboration with machine learning experts. IP6's breakthrough came when they went back and questioned whether their features were appropriate. They then found that the creation of new features led to good results with a simple algorithm. In their words, "We basically tried a whole bunch of Weka experimentation and different algorithms ... and nothing worked, so we decided that ... maybe we should explore the feature space."

IP6's experience points to a common experience divide. Even though they were collaborating with machine learning experts – researchers who had spent their professional careers studying and applying learning algorithms – they could not get their model to work until they revisited earlier steps. The learning experts could help choose and analyze the state-of-the-art algorithms, but were less helpful with features because feature generation involves expertise in the data domain. Machine learning experts are not always domain experts. I found in my studies that often building an accurate model is in many cases dependent on effectively applying domain expertise to gather data and generate features. Much like in the case of IP6, simple algorithms can often suffice if the data and features work well.

In another example, IP5 described a frustration with the fact that a feature and learning algorithm provided accurate models based on cross-validation tests, but the models failed to work well when embedded in a deployed application. After months of testing, they found that the root cause of this error was bias in the dataset. The model had overfit to a single person who had provided the bulk of their training data. The collection of more varied training data significantly improved the results. Here, revisiting the data collection step proved fruitful for building a model that generalized in practice.

As a final example, IP5 recounted a case where they changed their problem definition. They were building an activity recognition system and wanted to build a system that could disambiguate between a wide set of activities. In their experiments, they found that their model could not effectively disambiguate between two related activities. This result was because the underlying data sources were not sensitive enough to capture the differences between the activities. They discovered that they could design around the failure by changing their problem definition. They combined the two error-prone activities into a single higher level activity. This provided a more accurate model, and they were able make slight changes to their application to account for the difference.

In lab, I observed the importance of revisiting previous steps in the pipeline. I computed the performance of the current best model by taking snapshots of each participant's workspace. I then manually coded screen captures of the study tasks. I annotated the timeline with the current step of the pipeline the participant was working on. Table 3.2 visualizes performance

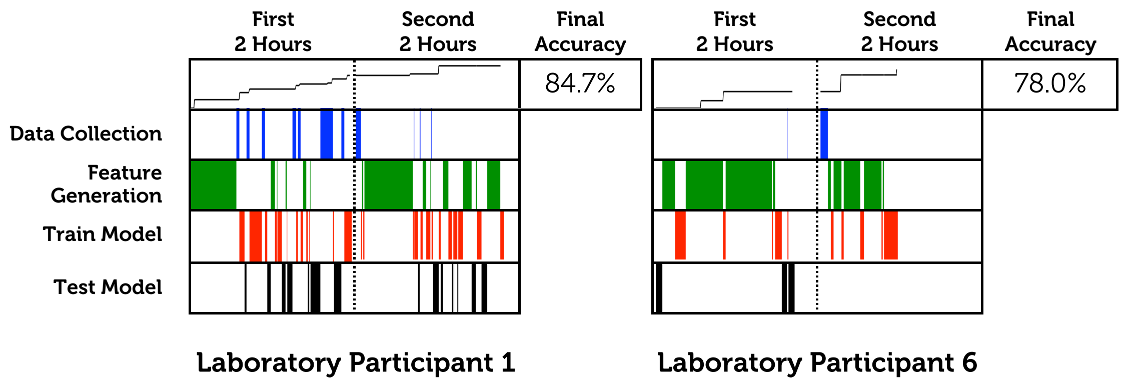


Table 3.2: A comparison of the processes of two representative laboratory participants. LP1 makes steady progress by iteratively exploring all portions of the problem, while LP6 spends that first portion of the study overly focused on feature generation.

and classification step over time for two representative lab participants.

LP1 revisits every component in the pipeline and makes steady progress, while L6 struggles in part because they become overly focused on picking the right classification algorithm. LP1 starts by investing some effort into feature generation, then begins a period of iteratively creating new training data, revising their feature generation code, considering different machine learning algorithms, and testing the resulting system. This iterative exploration can be seen in the very dashed nature of LP1's activities, as the participant explores all of the steps in the pipeline.

In contrast, LP6 spends almost the entire first half of the task focused on feature generation and makes relatively little progress, creating a model with one of the lowest midpoint accuracies. Once provided with additional training data (the 200 examples from four people that I provided each laboratory participant at the midpoint of the task), LP6's model is noticeably improved (and screen capture recordings show that LP6 made no significant changes to their features or their modeling algorithm). LP6 was unable to make this progress earlier at least in part because they were overly focused on feature generation and did not revisit their training data. L1's final model is a top performer, while LP6's final model performs average in comparison to other models.

3.4.2 Understanding the Relationships between Data and Code

To make progress on the pipeline, programmers need to be able to understand why their program is not working well. For training a model, this involves understanding the code, the properties of the data, and the relationships between data and code. Programmers inexperienced in using machine learning often don't know that they need to understand more than just the code. When they do realize that they need to look at more than just the code, current programming tools make it difficult for programmers to act on that knowledge.

Experts noted that many people who are new to building machine learning programs (sometimes the experts themselves at earlier points in their career) want to treat the programs as black boxes. Non-experts thought they could just provide data and the system would provide accurate models. They were not aware of the amount of work that went into collecting data, generating good features, and choosing appropriate evaluations. This section discusses four examples of the importance of understanding data, code, and the relationships between data and code to making machine learning algorithms work.

IP2 has a machine learning background and significant experience applying their background to human-computer interaction problems. In describing their general approach, they noted they initially focus on creating promising features. They create simple visualizations to see if there is any signal in the feature, and based on their familiarity with the algorithm they know if the algorithm will pick up on that signal and provide a good model. Familiarity allows experts to simulate the behavior of the model in their head.

After collecting appropriate data, IP2 iteratively creates and examines features. They generally do not apply a learning algorithm until after they are confident their features will work well. This approach represents one extreme. The expert feels that they understand machine learning algorithms to the point that they can simply inspect a set of features and know whether they will be able to train a good model.

IP1 describes another process where understanding features was important. In contrast to IP2, IP1 was not an expert, and as such they could not simulate the behavior of machine learning algorithms in their head. They describe a similar process of collecting data and generating feature (just like other participants). But when it comes time to use those features to train a model, they used hard coded rules rather than a learning algorithm to author a function.

IP1 prefers this approach because they know why their function behaves the way it does. If it behaves poorly, they can manually inspect their features and step through their heuristics to determine why the function failed. They can then address the failure by adding a new feature or a new heuristic. IP1 understood the tradeoff in accuracy; they knew that a learning algorithm might provide a more accurate model. However, they preferred the heuristics because they were more interpretable and allowed them to more easily connect the data, features, and results to debug the behavior of their function.

I observed similar preferences in my laboratory study. For example, LP9 started by using a decision tree algorithm, because it allowed them to easily see what features were being used and what relationships existed between features at different levels of the tree. The interpretably of the decision tree allowed LP9 to debug features – they would create new features and see which of them were the most informative according the decision tree. It was only when they had stopped working on features that LP9 started using more complicated algorithms to train accurate models. Their final model (a boosted ensemble of support vector machines) did indeed perform better than a decision tree, but it would be near impossible to understand the behavior of that model.

The previous examples demonstrate the importance of understanding the interactions between features and algorithms when debugging a model. But understanding is as important, perhaps even more so, when the system appears to be working correctly. IP3 recounts a case where trusting performance metrics for an opaque model was harmful. They had spent several months believing they had an effective model for classifying online forum posts. When training this model, they worked with a table of extracted features generated by another researcher. The performance metrics were high according to standard evaluation techniques.

However, IP3 never looked at the data or inspected the most informative features selected by the model when they were training the model. When they finally did, they discovered that the model was classifying data based on a specific feature. This feature was suspect because the relationship to the phenomena they were trying to model was tenuous at best. Digging into the data, they realized that the feature was picking up on an event that was coincidently correlated in their data set. When they had collected the data for a specific event they were trying to categorize, there had been a flood of spam messages on the forum post. Their model had picked up on the spam features, and when these features were removed the performance of the model degraded. It was only when IP3 looked at their data, features in conjunction with the classification results that they were able to find these errors in the dataset.

In the first three examples, participants were trying to figure out how data and features relate to experimental results. The processes were different but the goal was the same. Based on their understanding of data, code, and the interactions between data and code, these participants were able to figure out what to do next and train a better model. The final example shows the risks involved with not following this process. When the results of the model are divorced from the underlying data and features, it is possible to fall into a false sense of security. Programmers need to be principled and check their assumptions to make sure the data, features, algorithm, and evaluation criteria make sense.

Experts commented on the tendency of non-experts to treat algorithms like black boxes. IP10 warns against this, pointing out machine learning algorithms cannot do “semantic things” but that successful models instead require “really seeing what is happening with the data.” IP4 similarly cautions that if a person cannot understand at some level the differences between data, then it will be difficult or impossible to build a classifier that can.

3.4.3 Structuring and Tracking Experimentation

To understand how well a program is doing there must be metrics for evaluation. Evaluations produce results, and these results help guide experimentation. Choosing an evaluation metric can be a challenge in and of itself. Correctness is a standard metric. Computed values such as accuracy and F-score are proxies for correctness, as they measure various ways a model can or can not be correct. However, correctness is not the only metric. Other factors are also important when evaluating trained models.

IP11 discussed the need to balance the utility of a feature with potential privacy implications when using models based on personal data. They would have liked more accurate features and more fine-grained data, but more details about their users would have leaked potentially sensitive information. There “was kind of a tradeoff between what [they] would have wanted to have and what [they could] have.”

IP8 discussed the computational cost of feature generation for a document classification model, noting “If your document’s large, then it takes a lot of time.” and “Part of [making the algorithm faster] was to cut back on the features,” LP2 dealt with the same tradeoff. They used an algorithm to automatically generate a large number of features from their standard set of features. Feature computation was far too slow for interactive use, so they reduced their feature set by using a feature selection algorithm[1].

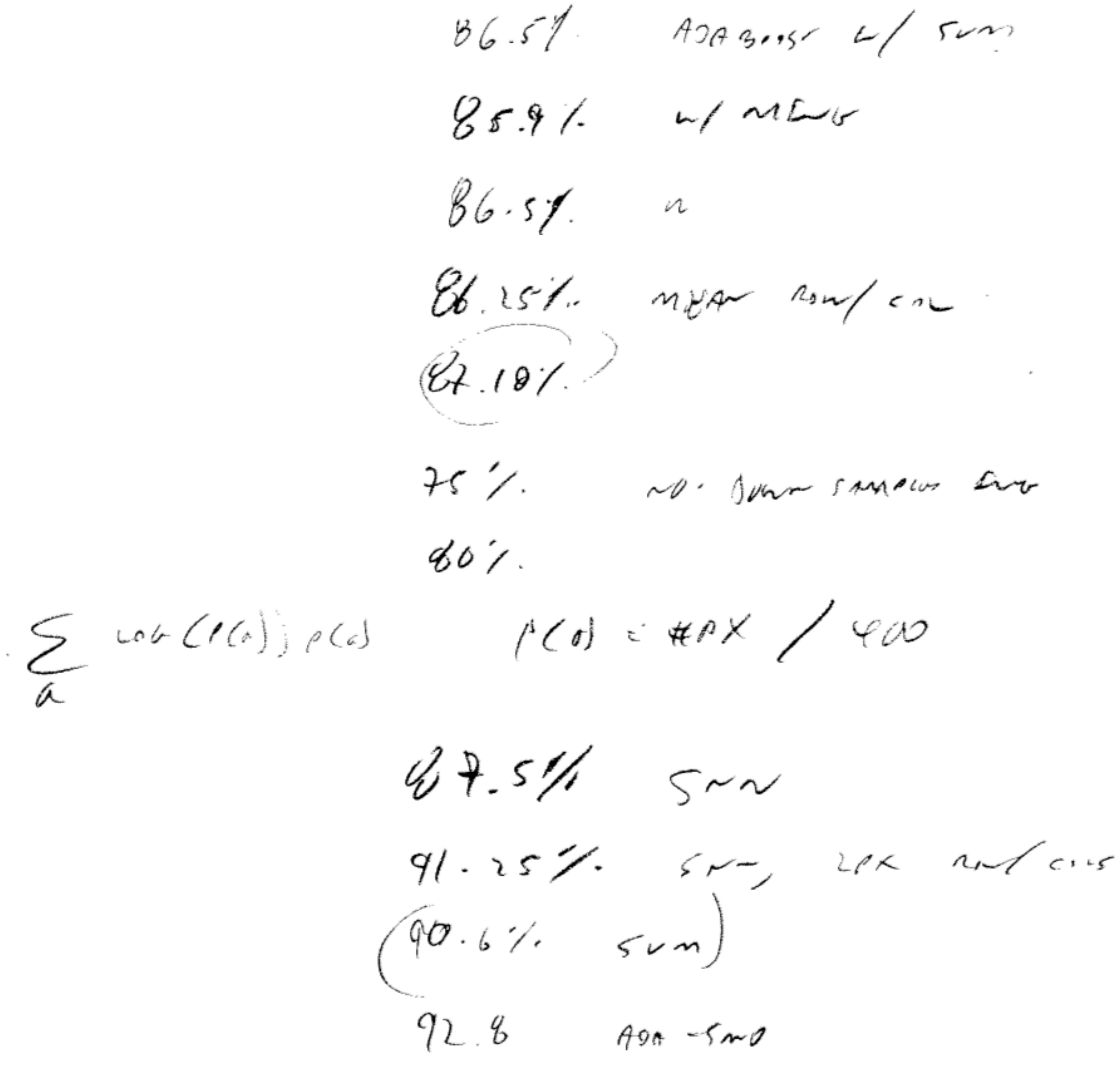
The point of experimentation is explore the space of possible models in order to train a model that performs well on the desired task. What performance means may vary based on the evaluation metric, but is often consistent across experiments. For example, if a programmer cares about accuracy, after every change in their program they can compute a new accuracy, compare the new accuracy with the old accuracy, and pick the program that performs better.

As they search through the space of possible models, programmers leave a trail of changes and experimental results. My studies show that tracking these changes and their associated results can help programmers converge on better models. For example, LP1, LP5, LP9, and LP10 kept logs of configurations they had tried and the accuracy they had obtained with those approaches. These four participants also trained some of the best models. Some of these participants kept paper logs with key parameters to the model and the accuracy of the model as seen in Figure 3.4. Others saved the model to file and encoded the important parameters into the filename (e.g., `LogitBoostWith8To18EvenWindow-Iter=10.model`).

While these participants did better in my short laboratory study, the experimental logging strategies they use are brittle and are likely to fail in the long term. Manual encodings are often incomplete. Programmers can't reproduce experimental results from incomplete records. These encodings also drift over time, so even if they were complete, comparing previous results in an experimental log to current results may be difficult. The importance of being able to go back and reproduce results becomes clear when there are bugs that affect the experimental results.

Errors that affect experimental results are common. For example, I found that programmers made mistakes experimentally testing accuracy, especially when working with data from people. At the core of this difficulty is assumption that data is independently and identically distributed (IID). Most standard tools provide a cross-validation test that makes this assumption; data is split *randomly* into folds and one fold is held out for testing while the rest are used for training.

[1]Interestingly, their decision to use a feature selection algorithm based on randomized 10-fold cross-validation using their entire dataset (as opposed to configuring a feature selection process to find features that work well across different people) probably led to significant overfitting and hurt their model's performance when tested against the 2000 new test digits.



86.5% ADABoost w/ SVM

85.9% w/ [illegible]

86.5% "

86.25% MEAN row/col

87.10%

75% no. Down samples [illegible]

80%

$\sum_a \log(P(a))\,;\,P(a)$ $P(a) = \#AX / 400$

87.5% SVM

91.25% SVM, 2PK row/cols

90.6% SVM

92.8 ADA-SMO

Figure 3.4: Some participants in my laboratory study kept paper logs of the algorithms tried and the corresponding accuracies. These logs can be useful – participants who kept the logs performed better than those that did not. But the logs are brittle because they are not complete. They still rely on the programmer to remember some of the decisions involved in generating the results (e.g., in the example above the programmer hasn't specified which dataset and feature sets they are using). If a programmer cannot remember these decisions, they cannot easily recreate the same experiments.

The random hold out creates problems when working with data from people. With models that must be robust across different people, the correct way to evaluate a model is to test it on people that *it has never seen before*. If the data is split randomly, both the training and testing data may have data from the same person. Because learning algorithms are fairly good at modeling data that they have already seen, the accuracy may be misleadingly high because the algorithm has overfit to the training data.

For example, in the digit recognition problem, the way I draw a four is consistent within my own data. If the training set has an example of one of my fours, it will do well at predicting the same types of fours in the testing set. Because my writing style is not necessarily consistent with someone else's writing style, this model has been tuned to my data and testing on my data will almost certainly lead to higher accuracy than testing on someone else's data. A handwritten digit recognizer should be robust to new data from new people, regardless of who it was trained on. The correct way to evaluate a model when working with people is to use a leave-one-out validation strategy. With leave-one-out validation programmers choose a person to exclude from the training set, train a model on the rest of the people, and test the model on the person that was excluded.

I observed this evaluation problem in my interviews. For example, IP5 recalled that "the cross-validation would show … 85% to 90% accuracy .. and then you would try it … it worked extremely well for some people and not well for others." By taking variation of people into account, IP4 was able to create a more accurate system. They reported a strategy of creating multiple models and adapting them based on the person. At the beginning of a deployment, IP4 tested to see which model seemed to be the best fit for each person in the deployment.

I tested these cross-validation errors in my laboratory study. Halfway through the experiment, participants were provided with more data from 4 different people. This addition of data was so they could test their assumptions and revise their model based on new data. There are a number of different ways participants could have incorporated this data. For example, participants might have trained their system using their own data and tested it against the provided data. Or they might have conducted leave-one-out validations, where they trained a model using three out of the four people, and tested against data from the remaining person. Either of these approaches would have likely provided some insight into how well their program generalized to data from new people.

Participant	Test Accuracy	2000 Digit Accuracy	Test Error	Number of Examples
LP1	87.8%	84.7%	03.1%	688
LP2	98.0%	75.3%	22.7%	200
LP3	89.1%	78.3%	10.8%	258
LP4	95.6%	82.9%	12.7%	250
LP5	91.3%	84.7%	06.6%	425
LP6	90.4%	78.0%	12.4%	230
LP7	72.0%	56.9%	15.1%	200
LP8	26.5%	22.8%	03.7%	200
LP9	92.8%	78.8%	14.0%	320
LP10	93.4%	84.4%	09.0%	500

Table 3.3: A comparison of how well participant's own tests indicated their models performed, how well they performed on 2000 new test digits, and how many training examples each participant used.

Instead, all participants trained a single dataset merging data from all four people (often adding more data of their own). They then evaluated the dataset using randomized 10-fold cross-validation. While randomized cross-validation is a standard technique for testing a system, it assumes that the IID assumption holds. Random cross-validation ignores the fact that the data was collected from four people. LP10 initially began to take an approach based in training with data from three people and testing against a fourth, but instead used randomized cross-validation because it was easier within Weka.

The left side of Table 3.3 shows one consequence of laboratory participants using misleading cross-validation methodologies. In comparison to evaluations that the participants performed, every model performs worse when tested against the final set of 2000 digits. A paired t-test indicates that participant estimates of how well their models performed are significantly higher than performance measures obtained using the 2000 test digits ($t(9) = 5.96, p < .001$). Such discrepancies can significantly impact a programmer's process: LP2, for example, quit the task with time remaining because their evaluations showed their model performing at 98.0% accuracy, though its performance on the 2000 digits was much lower.

In real world situations where programmers are trying to use trained models within applications, a false sense of accuracy is dangerous. When IP5 realized they had an error in their evaluation metric, they had to start the process of refining their data, features, and algorithm over again with their new evaluation metric. A common way to deal with leave-one-out cross-validation situations is to provide more data. With enough data, the effects of overfitting are diminished. State-of-the-art digit recognizers (like the ones used by the U.S. Postal Service) use massive datasets to avoid the IID assumption.

Evidence of this can also be seen by examining how many example instances each of laboratory participants used in relation to how closely their estimates of model accuracy matched final test accuracy (see the right side of Figure 3.3). An analysis of variance, excluding LP8 (whose failure to produce an effective model makes them an outlier for this analysis), shows that the number of training examples each laboratory participant used had a significant effect on how well their estimates of model performance corresponded to tests performed using the 2000 test digits ($F(1,7) = 14.00, p < .01$). While the collection of large datasets is a powerful approach, interview participants noted that the cost of such data collection can be prohibitive. For example, when asked whether they collected more data to further refine their model, IP10 responded "No. It was way too hard. There was no question."

Problems with the IID assumption are just one of many systemic errors that can invalidate past experimental results. Recall that IP3 had to restart the machine learning process from scratch when they learned about errors in their underlying data. Their evaluation criteria did not change, but they had to rethink their feature computation code and their algorithms. Everything they had done in the past was called into question.

This brings up an important observation. Because everything in the machine learning process is connected, there can be errors in the current pipeline that affect the integrity of past experimental results. Experimental results are how programmers make decisions about what works well and what doesn't work well, so changes in the integrity of past results bring to question the soundness of past decisions. For example, IP3 and IP5 can no longer assume the features that worked well in the past will work well in the future, and even more importantly they can no longer assume that what didn't work well in the past won't work well now. The fact that these errors occur relatively frequently is a reality that programmers must accept and work to overcome.

Experimental history is crucial for tackling these sorts of systemic errors. With good tracking, programmers can go back, apply bug fixes, and rerun key experiments. However, as my laboratory studies show, tracking experimental histories is hard to do manually. Reproducing and comparing results can be even harder with incomplete histories. Consequently, it is hard for programmers to check prior assumptions.

3.4.4 Breakdowns in Current Tool Support

With current machine learning tool chains, it is difficult to construct and iterate on a pipeline, understand relationships between data and code, and structure and track experimentation. One reason why these important tasks are difficult is that current machine learning tools are disconnected. The toolset used in my laboratory study is typical of what I observed in my interviews (Figure 3.3). Programmers have separate tools for collecting data, processing data into features, learning a model from featurized data, and testing the model.

Separate tools impose an application switching cost. To move between generating features in Java and choosing a new algorithm in Weka, programmers must save their data to an intermediate file format, quit Eclipse, switch to Weka, load data, and run experiments. Once in Weka, going back to generating features in Java requires another application switch. Interview participants reported overcoming this barrier by writing glue code to connect different steps in the classification pipeline.

Glue codes reduces application switching costs but is often brittle. Changes to the learning process lead to considerable changes in code. Additionally, programmers may lose important functionality. For example, if programmers use the Weka API to connect generating features in Java to choosing an algorithm with Weka, they lose much of the GUI support provided by the standalone Weka application.

The writing and maintenance costs of glue code are just some of the difficulties that programmers face. With current tools, information is lost when switching between data formats. For example, participants in the laboratory loaded the output of their feature generation code in Weka, but once within Weka they were unable to see the actual image of the digit associated with each feature set and results. This makes it hard to connect raw data (e.g., digit image) to features (e.g., pixel values) and experimental results (e.g., predicted labels). Connecting raw data to features and results is important for debugging the behavior of the model.

Experts prefer setting up their entire pipeline in a single programming environment like Matlab. Matlab allows programmers to better determine how best to proceed, because it allows them to better inspect their algorithms and data. IP10 stated "I think it's really valuable to work in an interactive environment, [because] you can go back and ask a data structure 'what did you do?' or you can add three lines and save off the state in some way." Similarly, IP10 said "If you have a black box that is a [machine learning algorithm] and it produces numbers in the end, then you have no idea what actually happened. So you need to be able to look inside the state of the algorithm and see what is happening, just like you would a program."

These tools work well for machine learning experts who know how to structure their code and can understand the inner workings of the algorithms. They are hard for non-experts to use. There is still a good deal of work that needs to go into connecting feature matrices and results in Matlab to raw data. In contrast to propagating the misconception that machine learning algorithms can be treated as black boxes, tools need to support the role of the programmer. Programers often have domain knowledge about data, the ability to collect data, and the ability to write code to generate new features. These actions allow programmers to embed their semantic understanding of the dataset and feature space into a form that the machine learning algorithm can understand.

My studies also show that programmers working on similar data generate the same types of features. Programmers could benefit from code reuse, but finding and integrating appropriate code is difficult [28]. A programmer's success in building a good model may lie not in their knowledge of the machine learning algorithm, their familiarity with the data domain, or even their knowledge about the existence of a informative feature. Success may hinge on their ability to integrate feature generation code to generate that feature within their code base.

This fact became obvious in my laboratory study. Participants used the internet to look for prior work on digit recognition and to look for code that implemented certain feature sets. In some cases, programmers tried to implement the same feature, but finding a working implementation was difficult. For example, LP4 and LP10 found discussions of the same feature when reviewing related work. LP4 was able to find an implementation of that feature and successfully incorporate it into their system, while LP10 spent time looking for implementations but eventually abandoned the feature. Current tools provide a common data structure for machine learning algorithms, but fail to do the same for feature generation algorithms. Programmers do not have a good way to find commonly used feature generation algorithms that work well for their data types.

In addition to difficulties training models, my participants had a hard time evaluating models in current tools. Tools like Weka put correctness forward as the most important evaluation criteria, but as my interviews show, correctness is just one of many evaluation criteria that programmers care about. Programmers need to look beyond accuracy and understand the performance of their model in the context of an application.

Finally, I observed that the experimental process is hard to track. Recall that programmers in my laboratory study that kept experimental journals did better than those that did not, but their methods for tracking performance (paper logs and filenames) were brittle. Current tools fail to help programmers track and log changes in data, code, and results. This failure can increase the effort needed to reflect on an experimentation history and re-execute prior experiments.

3.5 Implications for Tool Design

Disconnected tool chains are common, and in some cases unavoidable. For example, data collection may require programmers to leave their programming environment and go into the world to gather data. In this case, the cost of data collection is high and can't easily be reduced. However, in many cases the gaps between tools can be filled. For inspiration, we can look at the programming environments that experts use. In my interviews, participants found programming in Matlab to be particularly useful. It reduced application switching and data conversion costs by providing a central place for programmers to load data, generate features, choose an algorithm, run experiments, and visualize outputs.

However, tools like Matlab are still far from usable by non-experts. Many of the benefits that experts get from Matlab are based on expert knowledge of the process, the algorithm, and the structure of the pipeline. And experts must leave Matlab for certain types of analyses (e.g., connecting raw data to results). Expert knowledge about the structure of the pipeline as well as the exploratory nature of the machine learning process can be baked into tools. In Chapters 4 and 6 I present two new tools, Gestalt and Hindsight, that support programmers by supporting both the pipeline and the process. They bridge gaps between tools by providing programming environments in which programmers can generate features, choose a model, and run experiments by writing code and also explore data by visualizing data.

A good environment for exploring data helps programmers understand how their model works. Understanding how a model works allows programmers to make more informed decisions when they explore the space of possible models. New tools should be designed to encourage data analysis, especially understanding how data and code interact with each other. Most debugging tools focus on understanding code. Most visualization tools focus on presenting aggregate information based on common data types (e.g,. numeric data, nominal data, location data). Current tools do not have a good ways for programmers to easily write code and visualize data in the same environment.

A connected tool designed for machine learning can provide new visualizations that allow programmers to dig into classification results. For example, programmers may need to look at the data associated with specific cells in the confusion matrix to understand a confusion. Or they may need to look at raw data, features, and the results from experiments side-by-side to understand the relationships between code and data. Both Hindsight and Gestalt allow programmers to connect results to raw data, providing at best a visualization of that that data and at worst a link to the file from which the features were generated. In Prospect (chapter 5), I present a visualization tool designed to help programers understand how machine learning algorithms interact with raw data.

My studies show that tracking experimentation is difficult. Programmers are unable to keep detailed records of what they have tried in the past, and they have a hard time rerunning experiments when they find a bug in their data. These problems can be addressed through new tools that automatically track experimentation. These tools can provide visualizations that allow programmers to explore old experimental results, provide automated suggestions based on analyzing the experimental history, and provide new interactions that automatically run a series of experiments based on common tasks. For example, such a tool could respond to a bug in the experimental design by invalidating prior experiments and providing suggestions for modifying existing experiments so that results could be trusted. It could provide comparisons between old results and new results to help programmers revisit assumptions about the performance of prior features, datasets, and algorithms. Hindsight is an example of such a tool.

3.5.1 Limitations

My studies provide useful abstractions for understanding and supporting the process of training a model. For example, my interviews and laboratory studies point to the importance of exploration and experimentation to understanding the behavior of a trained model. Understanding helps inform the exploration of possible programs, and this is key to training good models. However, these are just the first of many studies that should explore this space. Additional work can expand the coverage of my results.

I interviewed computer science researchers. This population is ideal for an initial study because they are both early adopters of new technology and skilled programmers. However, there are many other populations that are looking to machine learning to help them solve their problems. For example, companies like Google and Microsoft use large scale machine learning systems to power their search infrastructure. Programmers at these companies will have difficulties understanding the behavior of their model and following an exploratory process. In contrast to our researchers, they may not have access to machine learning experts to pull them through difficult situations. Additionally their systems may need to be more robust, because they are being deployed in products. By studying different populations of programmers, future studies can provide a richer understanding of the machine learning process.

My laboratory studies look specifically at classification. These studies can be expanded to look at other machine learning techniques. For example, with clustering algorithms programmers do not have ground truth labels. In these situations, analysis may be more important. Programmers don't have a semantic understanding of what a cluster means. A semantic understanding is often necessary to use clusters effectively. Therefore, they may need to analyze the contents of the clusters in order to ensure that the clusters make sense.

My laboratory studies were conducted during one four hour session. This study's tasks were constructed to fit within this time frame. I provided structure that programmers normally wouldn't have. New studies should look at how different programmers construct a machine learning tool chain. These studies could analyze source code from existing open source machine learning projects, or they could observe students trying a model from scratch within a classroom setting. A longitudinal study could complement current observations by looking at how experimental tracking strategies help programmers recreate experiments weeks or months after they build an initial model.

3.6 Summary

In this chapter, I presented results from two studies that look at the difficulties programmers face when using machine learning to train a model. My studies of researchers allowed me to observe the process programmers take when using machine learning, and my laboratory studies provided grounded observations of the breakdowns in this process. Based on these studies, I found that programmers have a hard time structuring a pipeline, understanding the behavior of their model, and evaluating the performance of their model. I discuss breakdowns in current machine learning tools and provide design implications for new machine learning tools.

I found that classification is a popular and powerful machine learning technique. Building a classifier involves creating a data flow called the classification pipeline, which is a series of steps that transform raw data into a model. The classification pipeline is constructed through the machine learning process, which involves running experiments and analyzing results to explore the space of possible models. The next chapter presents Gestalt, a tool designed to help programmers write code to implement a classification pipeline and analyze data to understand the behavior of that pipeline.

Chapter 4 | GESTALT

Current general-purpose programming tools are not designed to deal with data, rather they focus on creating and managing code. Because the behavior of a model is based on both the data and the code, when training a model programmers need tools support working with both code and data. The traditional solution to this problem has been to constrain the data domain. Such constraints make training a model more accessible to the point that non-programmers can train models for these domains.

These domain-specific tools hide complexity in the interest of usability. For example, Crayons uses a coloring metaphor for training image segmentation functions [45, 46]. Users can easily provide the Crayons with data by coloring but they cannot change the feature generation or classification algorithms. Similar techniques have been used in a number other domains, such as sensor-based interaction and computer vision systems [54, 91].

The domain-specific nature of such tools is both a strength and a weakness. Domain knowledge allows tools to limit the decisions required for a programmer to train a model. But these same limitations also constrain the programmer if a tool's assumptions do not match the programmer's needs.

In this chapter, I ask the question: "How do we create a general-purpose programming tool that supports machine learning?" Such a tool would need to work across different data domains, provide support for writing code, and provide support for analyzing data.

As an exploration into the space of general-purpose programming tools for machine

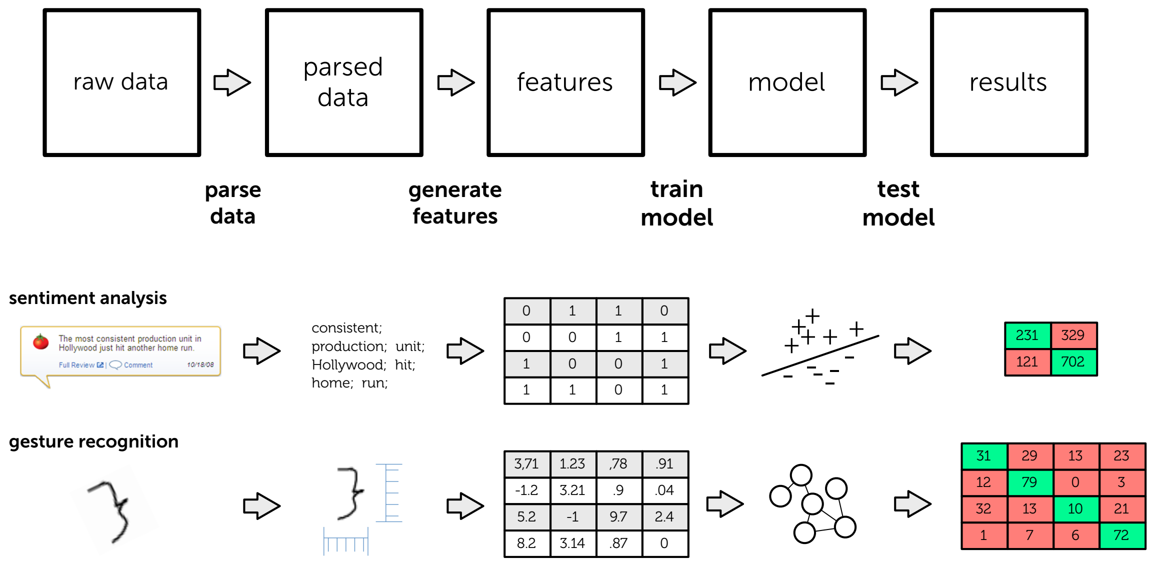


Figure 4.1: Both gesture recognition and sentiment analysis share a common high level data flow structure called the classification pipeline. Although the structure of the classification pipeline is the same, the logic of each step in the data flow is different.

learning, I have developed Gestalt. Gestalt helps programmers train models by supporting the entire machine learning process. Gestalt demonstrates new solutions to problems with current machine learning tool chains described in my previous studies. Specifically, it removes gaps between tools, allowing programmers to represent their classification pipeline and analyze data as it moves through that pipeline. In this chapter, I discuss how Gestalt builds upon findings from my studies to support the machine learning process, and I present results from a debugging study in which programmers using Gestalt found and fixed more bugs than with a state-of-the-art baseline tool.

4.1 The Machine Learning Process

In my previous studies, I observed that there is a structured process that programmers follow to train a model. I called this process the machine learning process. The machine learning process involves two high-level tasks: implementing a classification pipeline and analyzing data as it moves through that pipeline.

Implementation requires both the creation of a classification pipeline and collection of data to train and test that pipeline. Figure 4.1 shows two example pipelines, in which data is transformed into discrete examples, features are computed over each example, a classification algorithm is used to train a model, and the accuracy of that model is evaluated.

Not all pipelines are identical, but their structure is similar: a linear progression of computation that transforms data into a model that can be experimentally evaluated.

Analysis allows programmers to understand the behavior of a classification pipeline by examining how data moves through that pipeline. Beyond the correctness of any individual line of code, analysis requires developing an understanding of complex relationships between data, features, and model output. In addition to final model output, this requires examination of intermediate data to ensure that each step in the pipeline behaves as expected. Programmers examine whether data is correctly parsed and discretized, whether features are correctly computed, and whether the overall performance is sufficient for a problem.

Although the structure of a classification pipeline is linear, the process of implementing and analyzing it is not. Analysis of a current implementation informs a programmer's next implementation action. Programmers often revisit prior steps, such as collecting additional data, debugging implementation of features, brainstorming new features, or reconsidering their classification algorithm. The process of applying machine learning thus requires repeated transition between implementation and analysis. Gestalt is defined by supporting both implementation and analysis so that these transitions can be fast, fluid, and easy.

4.2 Providing General Purpose Support

This section introduces two canonical classification problems: movie review sentiment analysis and pen-based gesture recognition. I discuss important differences between these problems, as these differences illustrate a range of support needed in a general-purpose tool. I then discuss their similarity, as their common structure provides the basis for Gestalt's integrated support.

4.2.1 Two Canonical Problems

Sentiment analysis consists of categorizing text (e.g., movie reviews) according to some sentiment expressed in that text (e.g., whether a reviewer had a positive or negative impression of the movie). A canonical machine learning solution was developed by Pang et al. [107]. Following Pang et al.'s process, a programmer collects positive and negative movie reviews, formats reviews to plain text, and computes word-count features (the number of times the word appears in the review). They then prune words that are too common, too rare, or not descriptive. The resulting pipeline can be evaluated in a standard cross-validation experiment. This involves randomly splitting data into testing and training sets, creating models using the training sets, and evaluating the accuracy of those models on the test sets.

Pen-based gesture recognition is well studied, with Rubine providing a canonical approach [114]. A programmer collects strokes defined as sets of (x, y, t) triples, where x and y are 2D points and t is time. Because different people may draw the same gesture differently, data is typically collected from a large pool of people to help ensure trained models are robust to such variance. Strokes are normalized by rotating, translating, and scaling them to facilitate comparison. The normalized strokes are then used to compute features (e.g., the length of the stroke, measures of angles in the stroke). Cross-validation experiments then evaluate the pipeline.

4.2.2 Differences

Sentiment analysis is a two-class problem, whereas gesture recognition is multi-class. In the sentiment problem, classification errors are binary (i.e., reviews can be only positive or negative). In the gesture problem, it also matters how an example is misclassified. For example, it is important to know if rectangles are commonly misclassified as triangles. This added information can help a programmer identify the part of the pipeline responsible for that error.

These problems also differ in the visual representation of their data. Pen-based gestures have a natural and compact visual representation. A programmer can easily verify the label of a gesture by simply looking at a drawing of the stroke. In contrast, the sentiment of movie reviews requires significantly more time and effort to interpret. Movie reviews also human verifiable but require more attention than a gesture.

These problems also illustrate differing interpretability of their features, including verifiability and sparseness. Individual values of sentiment features are easier to verify. A programmer can quickly check the value of a word-count feature against the text of a review. In contrast, it is difficult to gauge the correctness of angle values and distances computed over the normalized points of a gesture. Additionally, sentiment features are sparse. Each review has a large number of features, most word-count values are zero, and only non-zero values have an effect on the final model. In contrast, the gesture recognition problem is defined by a small set of dense features, where each feature may have a distinct value and an effect on the trained model.

A final difference is how models are evaluated in cross-validation experiments. Random splitting of data into training and testing sets is generally effective for sentiment analysis and other problems. Applied to gesture recognition, however, it can be misleading. The way a person draws a gesture is often consistent for that individual, but can differ widely across people. Randomly sampling the dataset ignores this property. Consequently, data from an individual can be in both the training and testing sets. Because the goal is to evaluate how well a model is likely to generalize onto people who are not in the training set, leave-one-out cross-validation is instead a better choice. Models are trained with data from all but one person, then tested with data from that person.

4.2.3 Similarities

An important similarity is that the code for both sentiment analysis and gesture recognition has a similar high-level structure. This same structure is shared by many different classification problems. Although code to train a model is different at nearly every step, Figure 4.1 shows that they can both be represented as a classification pipeline. Both separate data into discrete examples, compute features describing each example, and conduct experiments that identify sets of examples that are correctly or incorrectly classified by the model.

This common structure provides leverage for a general-purpose tool. In the development of Gestalt, I examined how an integrated environment can provide necessary flexibility at every stage of a process while also leveraging this common structure to make programmers more effective when they train models.

As mentioned previously, the data for both problems is visual and examples are discrete. Visualizations for both problems are also easy to understand without domain expertise. In contrast, there are problems that are harder to visualize. For example, a programmer trying to recognize activity from sensor streams may need to combine many different streams and then run transforms that take those streams and project them onto some sort of frequency data space. Visualizations of the frequency space are hard to understand for programmers without experience in both signal processing and the data domain.

Additionally, both problems are easily solved using small datasets and simple algorithms. This means that programmers can train models and get feedback in real time. Real time feedback enables an interactive loop where programmers can make changes to their classification pipeline and quickly get results. In contrast, there are problems where large datasets or complex algorithms are needed to train a useful model. For example, training state-of-the-art video recognizers requires large datasets and a lot of computational resources [84].

The previous two similarities are important for my study. The ideas behind Gestalt can be scaled to situations where domain expertise is needed and where feedback is delayed (e.g., when feedback is delayed and when visualizing data is hard). However, for the purpose of running a laboratory study to evaluate Gestalt, I chose problems that are both interpretable and can be trained in real time.

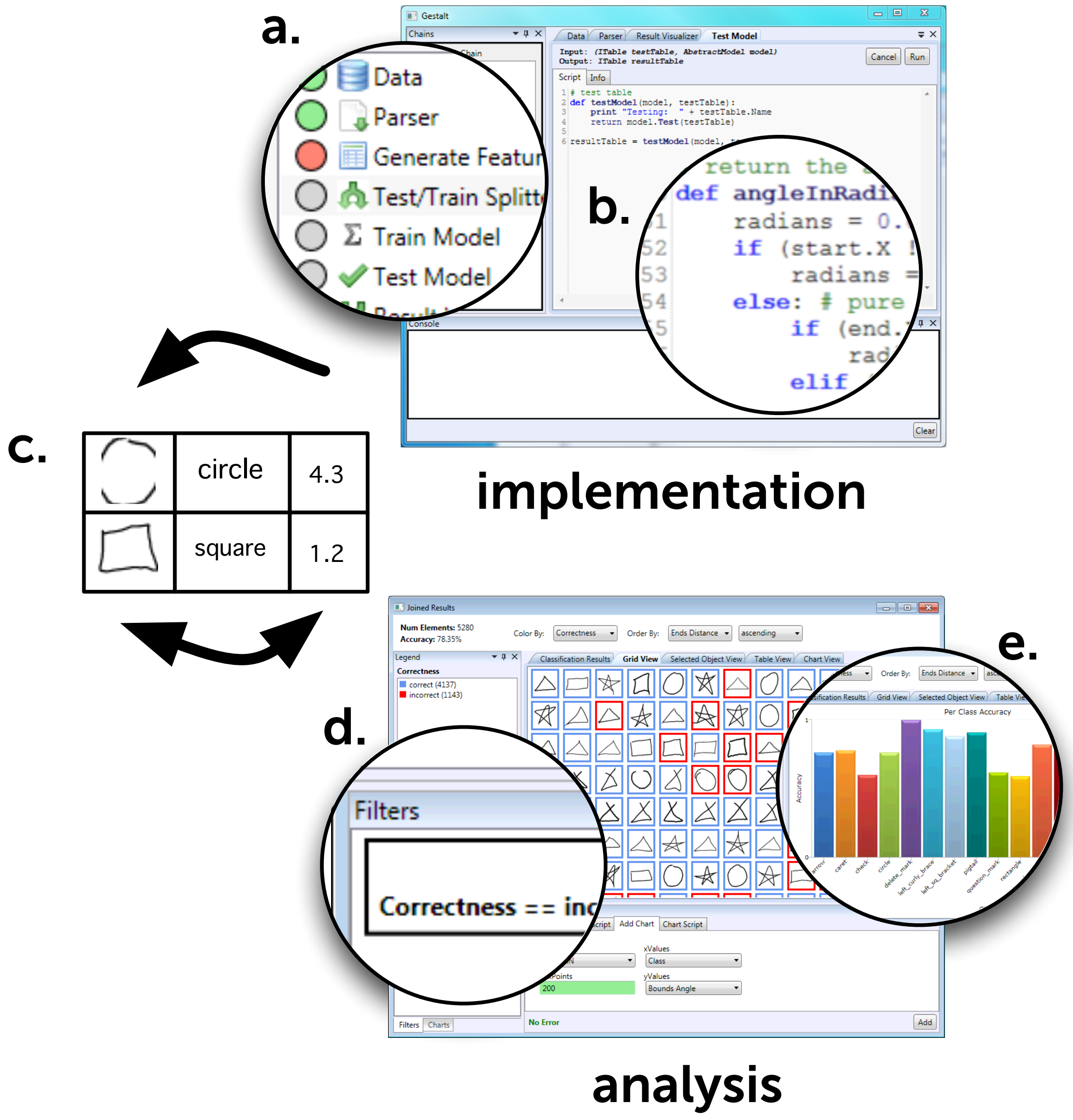
4.3 Gestalt

Programmers interact with a classification pipeline in Gestalt through two high-level perspectives: an implementation perspective and an analysis perspective (Figure 4.2). This parallels the common distinction between coding and debugging perspectives in modern development environments (e.g., Eclipse, Microsoft Visual Studio). The implementation perspective allows programmers to edit code and manage the classification pipeline. The analysis perspective visualizes the information computed as data moves through that pipeline. This section describes the specific capabilities of Gestalt and discusses how these capabilities work together to support programmers as they implement a pipeline, analyze data, and transition between these perspectives.

4.3.1 Providing Structure While Maintaining Flexibility *How do I represent my problem?*

Domain-specific tools use an understanding of a particular machine learning problem to constrain and hide some parts of the classification pipeline, exposing only some of the parts a programmer needs to interact with to train a model. For example, Crayons allows programmers to input data and see the output of a model, but provides no control over the features or the learning algorithm [45, 46]. Crayons achieves its ease of use by cloaking this complexity. However, reducing complexity comes at the cost of flexibility. For example, it is impossible to directly modify Crayons to solve a different machine learning problem, even if that problem has a similar classification pipeline.

A key realization in the development of Gestalt was that general support cannot be achieved



```
Input:  (ITable testTable, AbstractModel model)
Output: ITable resultTable
```

```
# test table
def testModel(model, testTable):
    print "Testing:  " + testTable.Name
    return model.Test(testTable)

resultTable = testModel(model, t
```

```
  return the a
def angleInRadi
    radians = 0.
    if (start.X !
        radians =
    else: # pure
        if (end.
            rad
        elif
```

Num Elements: 5280
Accuracy: 78.35%
Color By: Correctness
Order By: Ends Distance, ascending

Correctness
- 🟦 correct (4137)
- 🟥 incorrect (1143)

xValues: Class
yValues: Bounds Angle
Points: 200
No Error

Figure 4.2: The implementation perspective provides programmers with structure through its classification pipeline view (a) and flexibility by allowing them to write code to represent their specific problem (b). A common data structure (c), shared between analysis and implementation, allows programmers to quickly switch between the two tasks. The analysis perspective allows programmers to interact with the provided visualizations (e) by filtering, sorting, and coloring (d).

by hiding steps in the pipeline. The classification pipeline is similar for many problems, but the relative importance of different steps varies from problem to problem. Gestalt provides general support through a structured set of explicit steps with standardized inputs and outputs (Figure 4.2a). Gestalt preserves flexibility by defining each step using IronPython scripts written in a built-in text editor (Figure 4.2b). This combination provides an explicit structure without constraining what a programmer can do within that structure. Gestalt thus provides the same flexibility as general-purpose programming environments (e.g., Eclipse, Matlab).

Gestalt's explicit structure provides a basis for its other functionality. For example, Figure 4.2a shows how Gestalt can help programmers locate execution errors within specific steps. A circle next to each step is colored grey, yellow, green, or red according to whether the step still needs to be executed, is currently being executed, was executed successfully, or failed due to an execution error. The structured and typed sequence of steps also allows Gestalt to capture and visualize computation at intermediate steps throughout the pipeline. Capturing intermediate data provides standard memo-izing benefits [92]. In the context of training a model, intermediate data also provides additional analysis benefits. Each step can be used as a launching point for analysis, helping programmers better understand the behavior of their model through inspection of the input and output at each step.

4.3.2 Appropriate Data Structure
Where do I store my data?

Implementing a classification pipeline requires loading data and storing it in some representation for use throughout the remainder of the pipeline. Domain-specific tools can hide the details of data storage and management, but these decisions cannot be hidden in general-purpose tools. Data comes in many forms and sizes, so effective data management is a requirement for general-purpose tools.

Gestalt stores all information from the entire classification pipeline in a relational data table. Relational tables are a natural representation for discrete examples with many features. Because of this, they are also the backbone of many other general-purpose tools (e.g., Weka, Tableau). Gestalt differs from such tools because they do not address the entire classification pipeline (e.g., Weka focuses on a library of machine learning algorithms, and Tableau focuses on powerful visualizations of tabular data). Despite their common tabular nature, data representations in such tools are not identical. Programmers using combinations of tools to address an entire pipeline must therefore explicitly attend to format conversion. The narrowed focus of each tool also means that information is often lost or unavailable when converting between tools. For example, Weka and other tools that represent examples as vectors of features generally lack support for examining the original data used to compute those features[1].

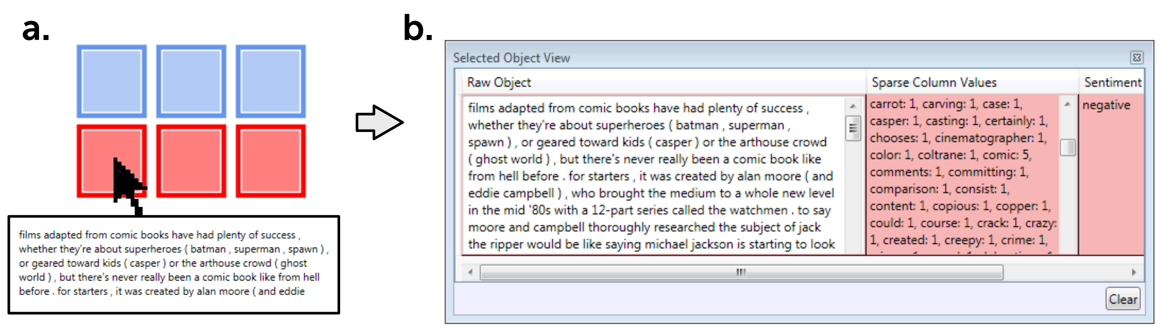
Gestalt's use of a single unified table means programmers are freed from managing data conversion or moving data between tools. This freedom is critical to enabling fluid and easy movement between interpretation and analysis. Gestalt's data representation also implements several enhancements to a standard table. First, feature columns are typed and tagged according to where they are used in the classification pipeline. All features can be used to summarize, visualize, and interact with data, but only some of those features can be used by a classification algorithm to train a model. Tagging of columns allows programmers to specify and Gestalt to track which features should be used by a classification algorithm to train a model. Instead of creating a separate data table in each step of the pipeline, Gestalt uses a cascaded table structure to reduce the overhead of storing intermediate data. Finally, Gestalt provides a sparse representation for storing large sets of sparse features found in many problems (e.g., sentiment analysis).

4.3.3 Visualizing and Aggregating Examples
How do I see my data?

Programmers reason about model behavior by examining data and its relationship to features and classification results. Domain-specific tools generally include a visual component that provides this feedback. This visual component allows programmers to examine individual examples as well as compare multiple examples. For example, Crayons presents images with translucent highlights indicating how pixels are classified by a trained model. This shows how individual examples are classified (individual pixels) and also provides relevant examples for comparison (the other pixels in the image).

Gestalt's support for many data types is enabled by a key distinction between individual and aggregate visualizations. It is impossible for a general tool to provide pre-packaged visualizations for all possible types of data. Gestalt instead supports data visualization by

[1]The raw data is typically not propagated forward by the feature generation script.



films adapted from comic books have had plenty of success , whether they're about superheroes (batman , superman , spawn) , or geared toward kids (casper) or the arthouse crowd (ghost world) , but there's never really been a comic book like from hell before . for starters , it was created by alan moore (and eddie

Raw Object	Sparse Column Values	Sentiment
films adapted from comic books have had plenty of success , whether they're about superheroes (batman , superman , spawn) , or geared toward kids (casper) or the arthouse crowd (ghost world) , but there's never really been a comic book like from hell before . for starters , it was created by alan moore (and eddie campbell) , who brought the medium to a whole new level in the mid '80s with a 12-part series called the watchmen . to say moore and campbell thoroughly researched the subject of jack the ripper would be like saying michael jackson is starting to look	carrot: 1, carving: 1, case: 1, casper: 1, casting: 1, certainly: 1, chooses: 1, cinematographer: 1, color: 1, coltrane: 1, comic: 5, comments: 1, committing: 1, comparison: 1, consist: 1, content: 1, copious: 1, copper: 1, could: 1, course: 1, crack: 1, crazy: 1, created: 1, creepy: 1, crime: 1,	negative

Figure 4.3: By looking at the raw data next to the features computed from that data, programmers can better understand the behavior of their model. Here a programmer is shown a thumbnail of movie review data (a). The programmer clicks on the thumbnail to examine the raw data, features computed from it, and the fact that it is currently misclassified (b).

separating the logic needed to view one example from the logic to combine many single examples into an aggregate view. Programmers can write code to visualize an example, and Gestalt then integrates it into aggregate visualizations throughout the pipeline. Two examples of aggregate visualizations are the grid view (Figure 4.3a, 4.4a, 4.4c) and the table view (Figure 4.3b).

Note that aggregate views begin to demonstrate how Gestalt's capabilities work together to create an integrated environment. Gestalt's structured representation of the classification pipeline defines boundaries between steps where programmers can use aggregate views to gain insight into their data. Gestalt's emphasis on code-based flexibility allows programmers to adapt those visualizations to meet the needs of their particular data.

4.3.4 Interactive, Connected Visualizations
How can I relate my data, feature and results?

Classification datasets are composed of discrete examples (e.g., gestures or documents) with associated ground truth labels. Experiments provide predicted labels for examples. Grouping and summarizing examples can help a programmer understand a classification pipeline. Gestalt's analysis perspective emphasizes interactive visualizations, inspired by work in interactive visualization tools [126]. Support is provided for faceted browsing, filtering, sorting, and coloring examples. Grouping and summarization operations can be applied according to feature values, according to columns added to examples by steps in the classification pipeline, and according to tags added to examples by a programmer.

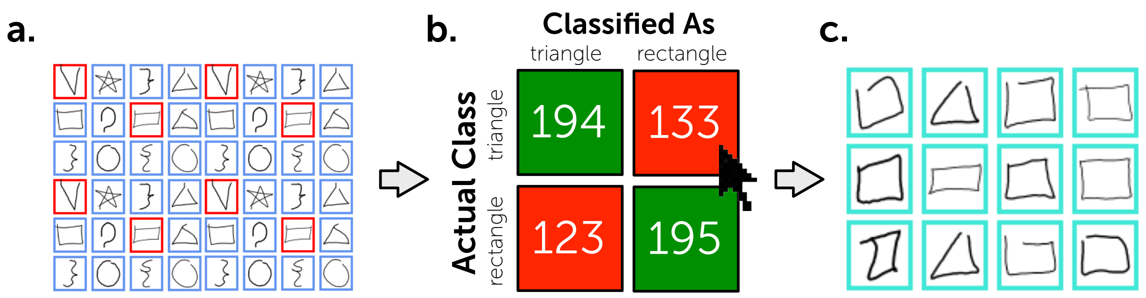


Figure 4.4: In Gestalt, programmers can use faceted browsing techniques to understand data. Here, a programmer tries to understand why triangles are confused with rectangles by filtering the full set of examples (a) through a click on a confusion matrix cell (b). The filtered examples (c) show that the confusion is due to mislabeled data.

Gestalt's support for machine learning goes beyond prior general-purpose visualization tools by connecting data generated across the entire classification pipeline. The coloring metaphor in Crayons is effective in part because it connects the pipeline's beginning (labeling data) and end (analyzing model classification) within a single visualization. Gestalt generalizes this with visualizations that connect data from different steps in the pipeline to help programmers understand relationships between data, features, and results.

Figure 4.3 shows one approach to a connected visualization: side-by-side presentation of information about the same example from different parts of the pipeline. Working on a sentiment analysis problem, a programmer hovers over an item in a grid view to see a preview of the document. They then click into the grid for a side-by-side view of the document, its computed features, and its predicted label. Pulling this into a single view allows a programmer to understand how an example moved through the pipeline.

A second approach to connected visualizations emphasizes filtering and grouping examples based on information from different steps in the pipeline. Figure 4.4 presents an example of a programmer clicking into a confusion matrix to isolate examples labeled as triangles and classified as rectangles. In this case, it seems likely that several of these instances are mislabeled. As another example, a programmer might apply a filter to isolate examples that have a particular feature value. Examining these might suggest a potential bug in the code that computes the feature. Connected visualizations allow programmers to quickly assemble the information needed to examine such questions.

4.3.5 The "Gestalt" of Gestalt

Each of Gestalt's capabilities is important, but Gestalt's real power comes from how they relate and are combined. Figure 4.4's clicking into a confusion matrix to see misclassified examples requires a structured understanding of the pipeline, the flexibility to implement an appropriate visualization of the individual examples, and a data representation capturing how each example moved through the pipeline. All of these pieces work together.

As a whole, these capabilities serve to accelerate the interactive loop of the machine learning process: programmers can more quickly implement and analyze different potential versions of a trained model. Gestalt's approach provides both structure and flexibility for rapid implementation, the shared data table removes data conversion and management to make it easy to switch between implementation and analysis, and connected visualizations allow programmers to quickly analyze the important parts of their classification pipeline.

4.4 Evaluating Bug Finding in Gestalt

My study compared debugging performance for participants using Gestalt with a baseline condition similar to Matlab. Recall that my interviews found that machine learning experts find that Matlab and other mathematical computing environments do the best job of supporting the machine learning process. This section describes the baseline tool and the method for the study.

4.4.1 Why Matlab?

In my interviews, machine learning experts preferred Matlab. Matlab supports the machine learning process better than most programming environments. Matrices are first-class objects, a good fit for tabular data representations. Many machine learning algorithms include solving linear algebra problems, also well-supported by Matlab. Matlab makes analysis easier by reducing the need to write boilerplate code needed to sort, filter, and create basic visualizations. Finally, Matlab provides sufficient functionality to reduce the overhead of switching between applications and connecting information across tools.

Despite these advantages of a connected environment like Matlab, it still falls short in addressing the difficulties programmers face when training a model using machine learning. Programmers must still construct a classification pipeline from scratch, as the environment does not understand the structure of the problem being solved. Matlab's data representation

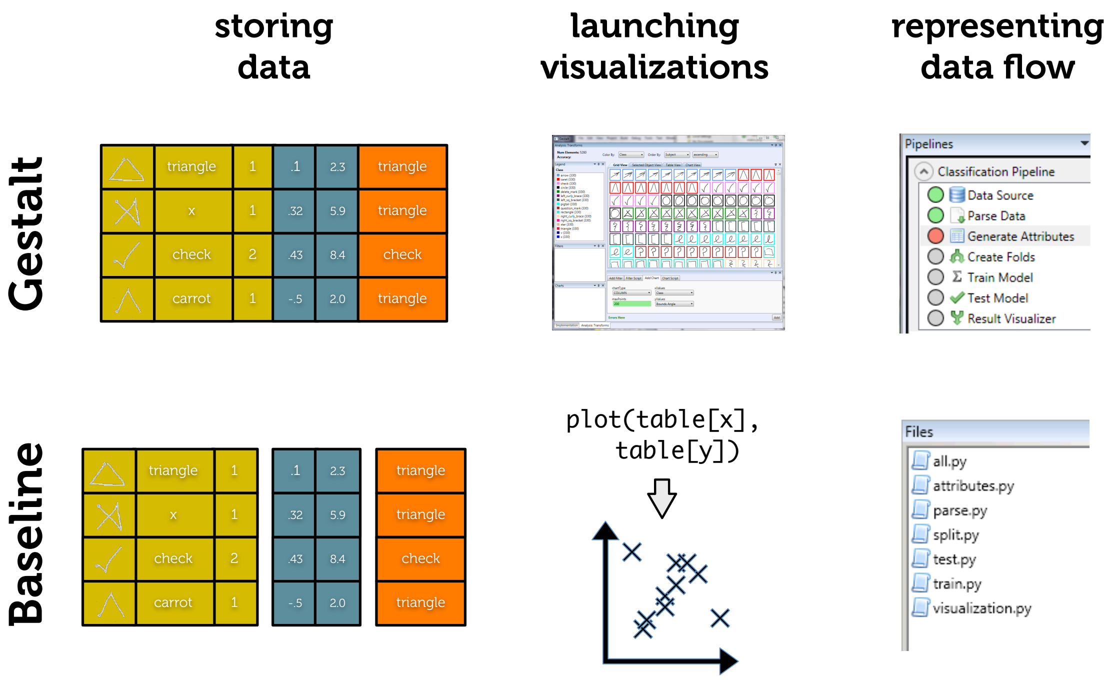


Figure 4.5: The baseline condition is different than Gestalt in three ways: data tables are not connected across different steps in the pipeline, visualizations are created using scripts rather than an interactive analysis perspective, and data flow is represented in files rather than a classification pipeline perspective.

has not been designed for machine learning, and all elements in a matrix are of a single datatype. Programmers therefore must maintain multiple parallel matrices to store raw data, numerical features, nominal (i.e., string) features, and feature names. Finally, Matlab visualizations are simple charts. They do not support the aggregation or visualization of raw data, interactively grouping examples within visualizations, or connecting information between different steps in the machine learning process. To support any of these capabilities, programmers would need to rewrite most of the functionality provided by Gestalt within Matlab.

4.4.2 Baseline vs. Gestalt

The baseline condition was a general-purpose development environment in which participants created, edited, and executed scripts. Like in Matlab, participants created visualizations by calling functions and writing scripts to sort, filter, and color. I provided an API with which could be used to reproduce all of Gestalt's visualizations. Figure 4.5 shows a breakdown the differences between the baseline and Gestalt.

The baseline condition and Gestalt used the same data table structure to store data. Unlike Gestalt, the data table in the baseline did not keep track of information generated across the pipeline (Figure 4.5 left). Participants had to write code to connect raw data, feature values, and classification results or to create side-by-side visualizations.

The baseline condition also didn't have the interactive visualization capabilities of Gestalt (Figure 4.5 middle). Programmers had to write code to filter, sort, and color examples. This is consistent with how visualizations are accessed in Matlab. I provided instructions and sample code for all of the visualization functionality of Gestalt.

The final difference was in how the baseline tool and Gestalt organized code (Figure 4.5 right). In Gestalt the code is organized and accessed through the representation of the classification pipeline. In the baseline the code is organized in files sorted alphabetically, similar to most programming environments. The decomposition of the data flow is the same, there is a file in the baseline that corresponds to each step in the Gestalt's classification pipeline, and there is an `all.py` file that executes the files sequentially.

Other than these differences, Gestalt and the baseline were identical. The entire process was integrated, all of the code for the learning process was written within the same framework, using the same data structures, with the same programming language. I chose this study design, instead of a design that compared Gestalt directly to Matlab, because I wanted to increase my confidence that observed differences were due to the functionality of Gestalt and not other differences, such as the syntax of the programming language.

4.4.3 Participants

I recruited 8 participants (6 male, 2 female) for the study. All were computer science graduate students. All had some experience programming in Python, had taken at least one course that taught machine learning algorithms, and had worked on at least one project that used classification. This population is consistent with the target audience of Gestalt: programmers with some prior experience in machine learning.

4.4.4 Study Design

The study was a within-subjects design, comparing Gestalt with the baseline across two debugging tasks. To account for carryover or interaction effects based on the ordering of interface conditions (i.e., ordering or pairing of interface and task), I counterbalanced the task with condition (Gestalt and baseline) and order (first and second).

The dependent measures included the number of bugs found and the number of bugs fixed within the one-hour time span of each task. A bug was counted as found if the participant verbalized the root cause. For example, "The data is mislabeled" or "This line of code should be using this variable instead". If the participant just speculated about the cause, the bug would not be counted as found.

I did not measure time to fix a bug, because it was not feasible to ascertain which bug a participant was working on at any given time. Participants were cognizant of the existence of multiple bugs. While trying to find and fix a primary bug, participants often gathered information needed to find and fix other bugs. Instead of the time to fix each bug, I focus on such measurements as the time spent in various visualizations over the entire study.

4.4.5 Sentiment Analysis and Gesture Recognition Tasks

Participants debugged solutions for the two problems discussed earlier: sentiment analysis and gesture recognition. Each contained `data` and five scripts: `parsing`, `features`, `splitting`, `training`, and `testing`. I created working solutions for both Gestalt and the baseline, then injected five bugs into each solution. Bugs were limited to the `data` and the first three files (`parsing`, `features`, and `splitting`).

The code for the baseline and Gestalt differed only in how scripts were called and how data was maintained between steps. Participants accessed the scripts by double-clicking on file names in the baseline conditions and double-clicking on steps in the pipeline in the baseline condition. These factors were intrinsic to the differences measured in the results. Some of the properties of the problems were described previously; here, I elaborate on those descriptions and provide details about their implementation.

The sentiment analysis task classified movie reviews as positive or negative. I used 1,000 negative and 1,000 positive reviews from a standard sentiment analysis dataset [107]. I computed word-count features, built a Naïve Bayes model, and evaluated the model using three-fold cross-validation. After creating a working classification pipeline, I introduced the bugs into the sentiment analysis problems. The following are the bugs that I introduced along with the associated step in the classification pipeline.

S1 The labels for 300 positive and 300 negative examples were swapped (in `data`).

S2 An extra for loop was added so that positive examples were read in twice (in `parsing`).

S3 Removing stop words (e.g., common words such as "the" and "a") is a common processing step in feature generation for text documents. A misplaced negation operator was inserted so that instead of removing stop words the code removes everything except for stop words (in `features`).

S4 In the loop that iterates over the features, the counter that moves through feature indices is not incremented. This leads updates for only one feature. (in `features`).

S5 The test sets are the same as the training sets (in `splitting`).

The gesture recognition task involved training a model that classifies a pen-stroke as one of 16 different gestures. I used a standard dataset of 5280 gestures collected from 11 different people [138]. I normalized strokes, computed features, built a Rubine model, and evaluated using per-person cross-validation. I introduced the following bugs:

G1 The labels for some of the gestures were changed, such that 30 triangles were labeled as rectangles, 30 rectangles were labeled as triangles, 30 circles were labeled as stars, 30 stars were labeled as circles, 30 carets were labels as checks, and 30 checks were labeled as carets (in `data`).

G2 The XML files for the gesture data had data stored in the following order (x, y, t), but the code loaded the data in the wrong order (t, x, y) (in `parsing`).

G3 The code that loads examples skipped over some of the data so not all of the examples were loaded (in `parsing`).

G4 The code for computing sine and cosine values was the same for one of the features (in `features`).

G5 The cross-validation always tests on the same person regardless of the training set (in `splitting`).

I chose all of the bugs based on common programming errors or common machine learning process errors. For example, earlier versions of the Pang et al. dataset included problems with mislabeled data that were later discovered and reported [107]. The cross-validation bug in the gesture recognition task is the same one reported by Hodges and Pollack in their work [61]. Other bugs were based on common mistakes, such copy-paste errors [69].

Participants were told that (except for the actual training and testing of the model) there could be bugs at any step in the pipeline. This included bugs in the raw data. They were assured the structure of the pipeline was correct and the task was not one of feature generation or algorithm development. As a stopping condition, they were given a target accuracy range suggesting they had fixed all of the bugs. This was a realistic stopping criterion in the context of the task, debugging existing machine learning programs that were known to have achieved a certain level of accuracy in the past.

Data-labeling bugs in each task would have taken more time to fix than was allotted. To make fixing mislabeled data tractable, participants had to clearly state why examples were mislabeled (associate the mislabeling with bad data rather than a programming error). I then pointed them to a directory containing correctly labeled data.

Finally, because the inserted bugs interacted with each other, the accuracy of the model could increase or decrease erratically (even going above the target accuracy). This was a deliberate choice; erroneously high accuracy values may be more dangerous because they provide a false sense of success. Additionally, it can often be the case that an existing solution may have multiple bugs and reported accuracy itself may not be the best metric for debugging.

4.4.6 Procedure

After providing consent, participants completed a one-page survey detailing their prior machine learning and Python experience. The experimenter provided a document detailing the first task. Both tasks were presented as salvaging code written by another programmer. The document detailed the steps taken by the previous programmer, and participants were informed the programmer had chosen a good strategy but there were mistakes in the execution. After explaining the task, the experimenter provided participants a one-page questionnaire asking what tools they would normally use to implement the outlined task.

After completing the questionnaire, participants followed a tutorial on each tool. In the Gestalt condition, the tutorial discussed the capabilities of the implementation and analysis perspectives. The baseline tutorial contained information about the capabilities of the editor and the visualization API. After the tutorials, the experimenter provided quick reference sheets for the included APIs. Because I was studying the effect of Gestalt's novel capabilities and not the usability or learnability of the model, participants were told they could use the experimenter as an intelligent help system during the task. This included asking questions about APIs, visualizations, the classification problems, and error messages.

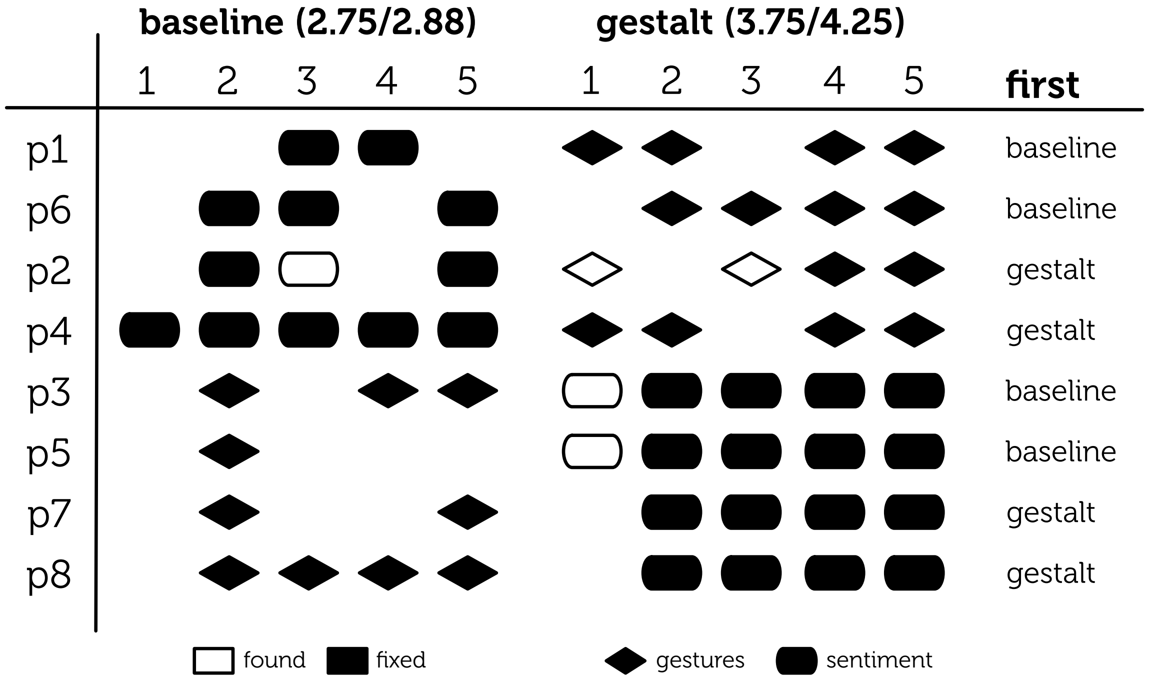


Figure 4.6: Programmers found and fixed significantly more bugs in the Gestalt condition.

Participants were asked to talk aloud, describing their progress in the debugging process. Participants were told the experimenter might ask questions about their state or current action. They were asked to choose among five distinct states: (1) I have no idea what the bug is, (2) I have a guess, (3) I'm checking my guess, (4) I'm fixing the bug, and (5) I'm confident I fixed the bug. Participants were given one hour to complete the task. After they finished, the experimenter saved their data and started the next task, providing descriptions of the new machine learning problem and the new development environment.

After completing the second task, participants were given a final questionnaire asking them to rate the usefulness of the visualizations and faceted search capabilities. They were also asked to compare the two development environments and to compare to the existing tools they had reported they would use for these tasks. Participants then completed a recording consent form and were paid $50 for their time. The entire study took between 3 and 3.5 hours.

4.4.7 Results

Participants unanimously preferred Gestalt and were able to find and fix more bugs using Gestalt than using the baseline. Figure 4.6 shows an overview of bugs per condition. To examine found and fixed measures, I conducted a mixed-model analysis of variance. I modeled participant as a random effect and modeled condition (Gestalt vs. baseline), task (sentiment analysis vs. gesture recognition), and trial (first vs. second) as fixed effects. I also modeled the interactions $condition \times trial$ and $condition \times task$. I used these same independent variables in all of the analyses reported in this section.

I found a marginal effect of trial on the number of bugs found, with participants finding more in the second trial (3.1 vs. 4.0 bugs, $F(1,5) = 4.62$, $p \approx .084$). This suggests some learning, as there were commonalities among the bugs in the two tasks. I verified the interaction $condition \times trial$ was not significant ($p > .42$), confirming the effectiveness of the counterbalanced design. Participants in the Gestalt condition thus found significantly more bugs (4.25 vs. 2.88 bugs, $F(1,5) = 11.42$, $p \approx .019$).

I also found a marginal effect of trial for bugs fixed (2.88 vs. 3.63 bugs, $F(1,5) = 4.09$, $p \approx .099$) and again confirmed the counterbalance effectiveness by verifying the lack of significant interaction $condition \times trial$ ($p > .72$). Participants in the Gestalt condition fixed significantly more bugs (3.75 vs. 2.75 bugs, $F(1,5) = 7.27$, $p \approx .042$).

4.5 Discussion

This section discusses how Gestalt was used, the process participants followed to solve machine learning problems, and possible explanations for Gestalt's better performance. The observations are grounded in answers to free response questions from the study questionnaire and in secondary measures of performance.

4.5.1 The Importance of Structure

I hypothesized a structured representation would be most useful when programmers first started a project, as it would be less daunting than a blank slate. Because I provided a mostly working implementation of the project, I felt the importance of structure would be diminished in the study. Consequently, I did not explicitly ask participants whether they found the structure helpful.

However, I included open-ended questions asking participants what capabilities they found the most useful. In this open-ended portion of the questionnaire, five of eight participants said the explicit structure provided by viewing and interacting with the classification pipeline was one of the most useful components of Gestalt. They stated they would like to see it in their own tools, with one participant writing: "The [classification pipeline view] was very helpful. When I am running these types of experiments, I often get lost in all of the processing steps. This seems like a useful way to organize the workflow."

Participants may have found the structure useful for a number of reasons. For example, as the previous participant notes, the structure offloads the mental effort needed to navigate a complex project and localize errors. If the participant suspected that there is an error in loading data, they could focus their attention on a specific part of the code. This effect may be magnified because participants were working with other people's code. Instead of spending time understanding the data flow by looking at model output and tracking code through files, participants could see the sequential data flow and the inputs and outputs to each step.

4.5.2 Creating Individual Example Visualizations

Even though I provided standard visualizations of the individual examples in both conditions, some participants tried to visualize examples on their own. Both the baseline and the Gestalt conditions provided developers with the ability to make charts, including the ability to plot points. Two participants in the baseline condition (P5 and P7) visualized gestures by using a scatter plot to plot the (x, y) coordinates of each point in the stroke. They did this because they did not trust the example-specific visualizations we had given them. Those visualizations looked wrong because (x, y) coordinates of the gestures were incorrectly parsed (bug G2).

Recall that Gestalt relies on programmers to create example-specific visualizations, which it aggregates to provide programmers dataset visualizations. One might ask, "is it reasonable to expect programmers to build example-specific visualizations?" These results suggest the answer is yes. Programmers need to see data to be able to debug the performance of their model, and they can and will develop quick, simple visualizations of raw data when given proper tools.

4.5.3 The Need for Connectivity

Participants in both conditions actively tried to relate feature values and results to their raw data. Gestalt's connected visualizations make it easy to compare data, features, and classification results. When taken away in the baseline, participants expressed frustration. One participant, who worked in Gestalt first, explicitly described that they wanted to see the raw data next to the features in the baseline and was annoyed that it was not as easy as in the prior condition.

To make up for a lack of connectivity in the baseline, three participants (P1, P3, and P8) went to great lengths to cobble together their own side-by-side table view. Participants added table visualization function calls at the end of the `parsing` and `features` scripts. They then resized the tables to look at the raw data and features side-by-side. Side-by-side visualizations seem to be a natural need for debugging machine learning programs, as two of the three participants created them before having used Gestalt and seeing its built in side-by-side table view.

4.5.4 Interactivity

I also observed that the interactivity of visualizations was critical. Because I logged the active window as well as input (e.g., mouse clicks, key strokes), I could determine if participants spent their time implementing or analyzing. Participants in Gestalt spent significantly more time analyzing instead of writing code (37.3% vs. 18.9%, $F(1,5) = 5.44$, $p \approx 0.001$).

Participants in the Gestalt condition also used more kinds of views. In the post study questionnaire, I asked participants to tell me which faceted search capabilities (e.g., filtering) and views they used (e.g., grid view). I found that participants tried significantly more views in the Gestalt condition (3.4 vs. 2.5 views, $F(1,5) = 18.84$, $p \approx .007$) and marginally more faceted search techniques (2.0 vs. 1.1 techniques, $F(1,5) = 5.44$, $p \approx 0.067$).

The gesture recognition task also led participants to spend more time in visualizations (32.9% vs. 23.3%, $F(1,5) = 11.15$, $p \approx .021$), look at more views (3.4 vs. 2.5 views, $F(1,5) = 18.84$, $p \approx .0074$), and use more faceted search techniques (2.3 vs. 0.9 techniques, $F(1,5) = 13.44$, $p \approx .015$) than the sentiment analysis task. This result is likely because there were more classes in the gesture condition and the data was easier to visualize. These differences suggest that spending more time looking at more kinds of views might allow programmers to better formulate and test possible explanations that lead them to find and fix more bugs.

In both conditions, most participants used filtering and sorting to group relevant examples. Gestalt made this easier. One participant followed the exact process shown in Figure 4.4. They clicked in a confusion matrix to see examples of triangles classified as rectangles, then found the mislabeled examples.

4.6 Limitations

My debugging study provides evidence that supporting the machine learning process in general purpose tools is not only possible but also useful for common programming tasks. However, there are still limitations of this approach. In this section, I discuss limitations of the study, Gestalt's implementation, and of general-purpose machine learning tools.

4.6.1 Study Limitations

My study has several limitations. Both tasks had classification pipelines that could be run in real-time (loading and processing data, generating features, training a model, testing the model). Many important learning problems are too expensive to be computed in real-time. I chose this limitation to allow participants to explore a large number of different bug hypotheses within the time constraint. It is possible that Gestalt may be more useful in situations where models take longer to train. Programmers might benefit more from using visualizations to explore data and features while waiting for updated results in a longer feedback cycle.

My study was also limited to finding bugs in unfamiliar code. Challenges in the middle of a development process are different from those at the beginning, and setting up a workflow for a learning task can be daunting. However, participants found value in Gestalt's classification pipeline structure, and their comments in the open-ended questionnaire lead me to believe Gestalt's structure will also assist programmers training a model from scratch.

My study focused on two problems for which programmers had some intuition about the data. This intuition allowed programmers to effectively tackle the debugging task. They knew gestures that looked similar should be in the same dataset, and they knew words in movie review text should appear as non-zero features. Programmers may not always have such a clear understanding of the data at the onset of the project. They may instead develop understanding over time. Flexible visualizations seem crucial for this, as they can allow programmers to create individual visualizations, presenting the information that will best help them to understand their data.

4.6.2 Limitations of Gestalt

My study revealed some unexpected work patterns that suggest new opportunities for Gestalt and other tools. Participants P7 and P8 created toy review datasets to see if reviews were being correctly parsed and word counts were being correctly computed. Participant P8 also created simple strokes that consisted of a few (x, y, t) points. They then manually computed features (using pen and paper) and compared them to the values computed during feature generation. Other participants created filters by manually selecting a small set of examples and examining them through the entire pipeline. These behaviors collectively suggest support for unit testing practices could be a good addition to Gestalt and other machine learning tools.

Altough Gestalt can be used to train models for many domains, there are some domains Gestalt does not completely support. A key limitation is that Gestalt assumes individual examples can be processed without the context of the larger dataset. This limitation impacts the types algorithms Gestalt supports, but also some of Gestalt's core capabilities. For example, the current grid and table aggregate visualizations cannot properly visualize relationships inherent to sequential data (e.g., time-series). It is also not obvious how to implement the interaction in Crayons, where individual pixels have meaning only in the aggregate context of an image. New general methods for describing relationships between examples would benefit Gestalt and future general-purpose tools.

The difficulty of implementing the core Crayons interaction within Gestalt raises a question of whether general-purpose tools can be as effective as domain-specific tools. Both styles of tool are important. It is almost certain that a highly-specialized tool will be more effective for its particular problem. However, general tools provide two advantages. I have noted that the number of domains affected by machine learning is large and growing. General tools can support problems for which domain-specific tools have not yet been developed. Further, distilling general mechanisms, like those in Gestalt, informs domain tools by allowing a focus on domain-specific extensions instead of re-inventing general mechanisms.

4.7 Summary

In this chapter, I presented Gestalt. I broke down the machine learning process described in Chapter 3 into two main tasks, *implementation* of a classification pipeline and *analysis* of the data and results. Gestalt is built to support implementation and analysis for classification. It provides structure for implementing a classification pipeline and interactive visualization tools for analyzing data. Because it is an integrated tool, Gestalt makes it easy to transition between implementation and analysis.

A debugging study of programmers using Gestalt versus a Matlab-like baseline shows that Gestalt helps programmers spend more time analyzing and less time iterating on code. The study also shows that programmers using Gestalt outperformed programmers using the baseline at both finding and fixing bugs.

Gestalt provides a framework for programming with machine learning. However, understanding relationships between data, features, and algorithms is still hard. In my studies, I observed that programmers spend a lot of time tweaking parameters to algorithms in order to get better performance. They are effectively exploring whether tweaking a parameter leads to a better model. Algorithms and parameters, or more generally code, are only part of what determines the behavior of a trained function. Programmers also need to understand data. In the next chapter, I present a new technique for understanding data and a tool called Prospect in which this technique is embedded.

Chapter 5 | **PROSPECT**

Programmers change parameters to algorithms and features to see if those changes have an effect on performance. Parameter exploration is easy, and people take the path of least resistance when attempting to solve problems [140]. Changing a parameter often involves simple actions, such as renaming a function call or clicking on a checkbox. In addition, parameter exploration is more natural for programmers than exploring data. Programmers are used to authoring the behavior of a function. If a function is incorrect it can be fixed by modifying code, in this case changing parameters. Consequently, for training behavior it is natural for programmers without machine learning expertise to ascribe poor performance to code rather than data.

Focusing solely on code doesn't always work for training behavior, because data is as important as code when training a model. My studies show that often simple algorithms work well. When a function is performing poorly, real progress often comes when programmers clean their data, collect better data, or generate descriptive features. Understanding the relationship between data and the behavior of the learned function is hard. To help programmers better understand data, I have created a new data analysis technique based on aggregating predictions from multiple models.

In this chapter, I discuss my approach in the context of a new interactive visualization tool called Prospect. I present two experiments that show how analysis using multiple models can help programers better understand data. The first experiment shows that using multiple models to identify potential label noise (i.e., mislabeled data) can provide a threefold reduction in the number of spurious examples a programmer examines. The second experiment shows that analyses of multiple models can identify examples that are significantly more likely to respond to additional informative features, thus helping a programmer focus their attention on the most relevant examples in a dataset. I conclude by running a user evaluation, which shows that programmers can effectively use Prospect to understand their own data.

5.1 Using Multiple Models

Prior work on visualizing the behavior of models has mostly looked at a single algorithm. Researchers have created visualizations that help programmers better understand the behavior of common algorithms such as decision trees, support vector machines, and Naïve Bayes classifiers [18, 22, 32]. While these techniques are useful, programmers are still left with the question: Is poor performance due to choosing a bad algorithm or using the wrong parameters?

Programmers try many different algorithms and parameters to answer that question. When evaluating the differences between resulting models, programmers don't always have a complete picture of the differences between models and the connections between those models and the underlying data. If a support vector machine performs better than a decision tree, there may still be examples that the decision tree consistently classifies correctly that the support vector machine does not. There may be examples that are always classified correctly by both algorithms. These example-level comparisons can lead to a better understanding of the data (e.g., examples that are always misclassified might shed some light on what is hard to classify in the dataset regardless of the algorithm).

Prospect supports deeper, example-centric comparisons by aggregating predictions from multiple models. These comparisons allow programmers to perform new queries on their dataset. For example, programmers can look at examples are are always classified incorrectly by all of the model. Prospect leverages the bias-plus-variance decomposition framework for evaluating classification algorithms [80]. This framework defines three values. Target noise is what we are trying to find in the data, the aspects of the data are currently too hard to distinguish. Target noise is inherent to data (i.e., it will exist even in perfect conditions). Programmers decrease the noise in the data by providing more structure through new descriptive features or more data.

Bias is the difference between the expected value and optimal (i.e., structural error of the model). Not all models will be able to fit the data. For example, if the data is generated by a polynomial function, a linear model will only be able to approximately fit the data. Bias provides some notion of how well a model can do in the best case.

Variance is how much an algorithm's predictions vary based on different training data. Changes to cross-validation folds can help test an algorithm's variance. It shows how susceptible classification results are to random noise. If models vary greatly on the dataset, they may perform poorly in real world conditions. To make a model more stable, a programmer may need to gather more descriptive features or collect more data to train a robust model. All models have some inherent bias and variance, but these differ between algorithms.

The idea that multiple models can be combined to useful effect is not new. Ensemble methods exploit intuitions about bias and variance to achieve better performance by combining results from multiple models. Combinations of classifier outputs based on simple rules (e.g., majority vote, sum) often produce results better than a single model [71]. Other ensemble techniques, such as Boosting [118] and Bagging [29], automatically generate simple models and combine them to build more accurate ensemble models. However, this work is generally aimed at learning ensembles to directly improve accuracy. My approach differs in that I leverage differences in biases to help programmers better understand data independent of the underlying algorithm.

5.2 The Prospect System

In order to make correct choices about which algorithm to use, a programmer must understand key properties inherent to the data. These properties are independent of any particular classification algorithm. Prospect lets programmers better understand key properties of their data by first training a collection of models and then analyzing the behavior and output of those models. The collection of classification models acts as a lens for scrutinizing some of the key properties of data. Figure 5.1 presents an overview of Prospect.

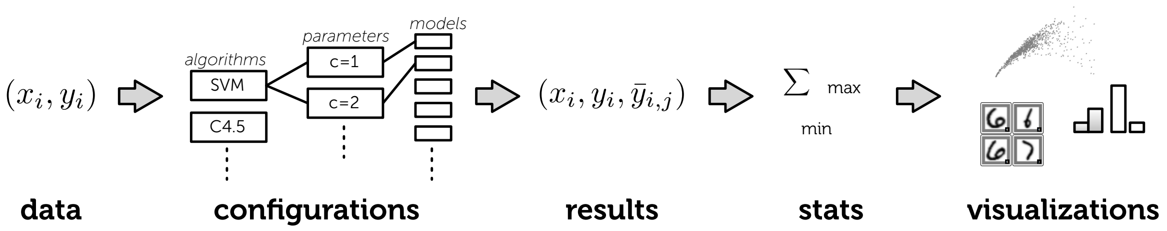


Figure 5.1: Programmers provide Prospect with the dataset they are trying to understand. Prospect trains multiple models (based on a set of configurations) to generate multiple predicted labels for each example in that dataset. Prospect then generates descriptive statistics about the examples by aggregating those predicted labels and provides visualizations to allow programmers to explore generated statistics.

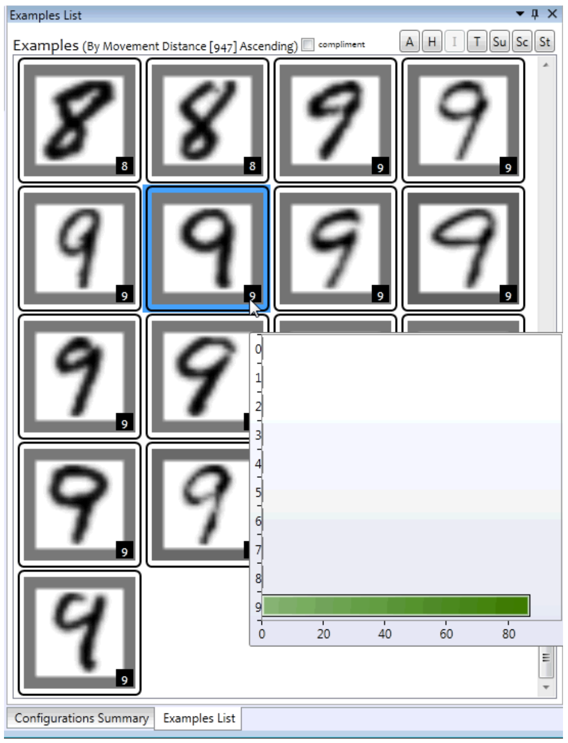
5.2.1 Processing Data and Generating Statistics

Starting from data, Prospect first generates a set of configurations. Each configuration defines feature selection procedures, a classification algorithm, its parameters, and other specifications needed to completely determine a model creation process. Configurations are generated by systematically varying algorithms and parameters throughout the process. The goal is to create configurations that can be used to generate a collection of models with different biases. Using the models in concert provides a new perspective on the data with lower bias. My intuition is that by training multiple models using a diverse set of configurations, Prospect can marginalize the individual bias of any particular model.

Prospect considers each available configuration and uses k-fold cross-validation to generate classification results for the entire dataset. Formally, data consists of a set of examples $X = \{x_1, \dots, x_N\}$, with labels $Y = \{y_1, \dots, y_N\}$, and C, a set of different configurations. I use $\overline{y}_{ij}$ to denote the predicted label of x_i resulting from the j^{th} configuration. Thus cross-validation for each configuration creates tuples of the form $(x_i, y_i, \overline{y}_{ij})$. Interaction with Prospect is based on summarizing these tuples.

One way of summarizing tuples is to group all of the tuples by configuration and compute statistic about each configuration. Programmers already do this. When using classification algorithms, programmers look at measures like accuracy, precision, and recall to get a better idea of how well a configuration is doing. Accuracy is computed by taking a single configuration and looking at the predictions for each example to see how many examples were predicted correctly.

Prospect's real power lies in its ability to provide new information about the data itself. In the



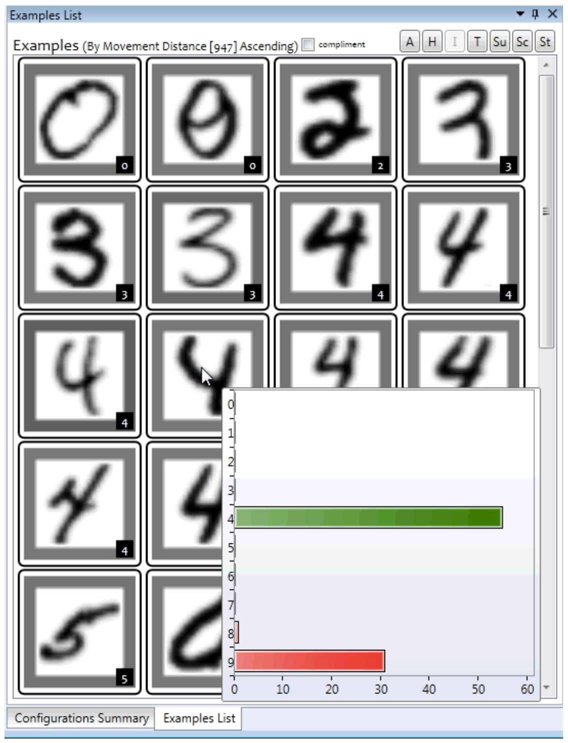


Figure 5.2: Prospect allows programmers to see the distribution of predicted labels for each example. Here the nine on the right is correctly classified by all of the configurations, the four on the right is misclassified some of the time. In some configurations it is classified as a nine. This information would be impossible to see by looking at the results for any given model.

case of systems based on a single model, the predicted label is the only new information about an example. In contrast, Prospect computes a distribution of predicted labels for each example. This distribution allows Prospect to compute example-centric statistics (e.g., the percentage of configurations that provide the correct predicted label, or the label that was most frequently predicted).

5.2.2 Interactive Visualization of Statistics

These statistics can then be used to visualize data. Prospect provides a library of common visualizations, such as histograms, scatter plots, and confusion matrices. Prospect provides example-centric visualizations of the data, such as the grid view. This view allows programmers to see examples in their dataset.

Prospect also provides new visualizations specific to multiple models. For example, in addition to looking at each example, programmers can see the distribution of predicted labels for each example. This allows programmers to see if an example was confused in *any* of the configurations. For example in Figure 5.2, the programmer can see that the nine is always classified correctly, whereas the second four is confused with a nine in some of the configurations.

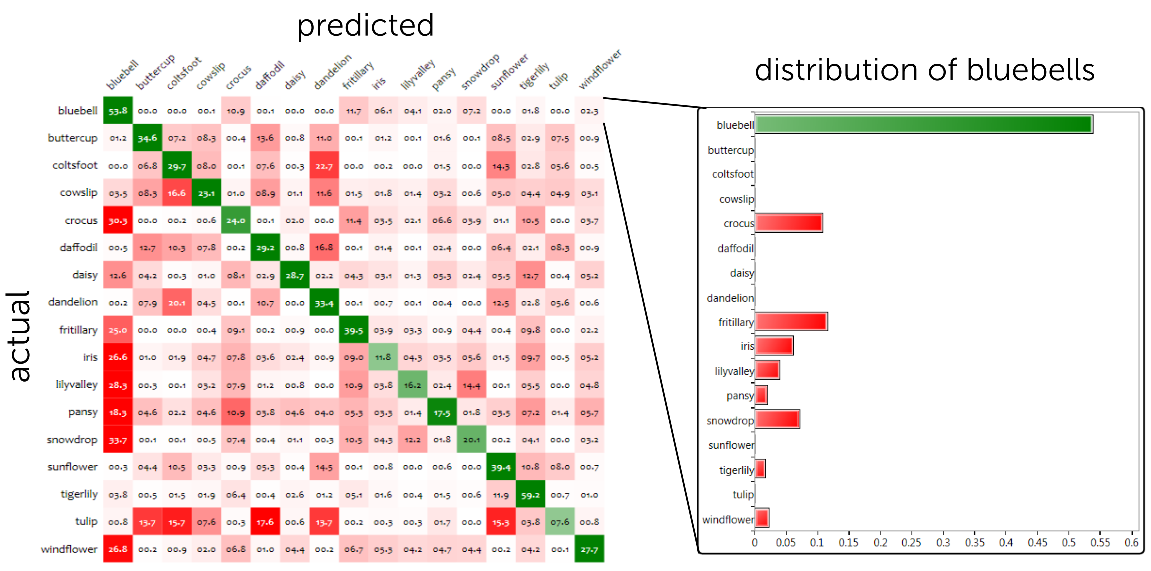


Figure 5.3: The aggregate confusion matrix groups examples by class much like a normal confusion matrix. Each row in the confusion matrix shows the distribution of predictions per class across all of the configurations.

Prospect also provides an aggregate confusion matrix (Figure 5.3). Recall that a normal confusion matrix breaks down predictions by ground truth class and predicted calls. High numbers off the diagonal indicate consistent misclassifications from one class to another. The aggregate confusion matrix takes this same concept and scales it to looking at multiple models. The matrix is created by adding together all of the confusion matrices and normalizing by rows. Each cell (x, y) in the aggregate matrix shows the percentage of time examples with ground truth x were predicted to be class y across all configurations.

Programmers can interact with the visualizations to filter their data. For example, they can select and filter examples on a scatter plot. Or they can choose to filter configurations by their accuracy value. Changes in filters modify computed statics. Modified statistics update the visualizations of examples. Instead of describing the full set of visualizations and interactions enabled by Prospect, the following sections focus on how the summary statistics and interactive visualizations can be used to address two problems: helping detect label noise and helping provide insight for generating new features.

5.3 Detecting Label Noise

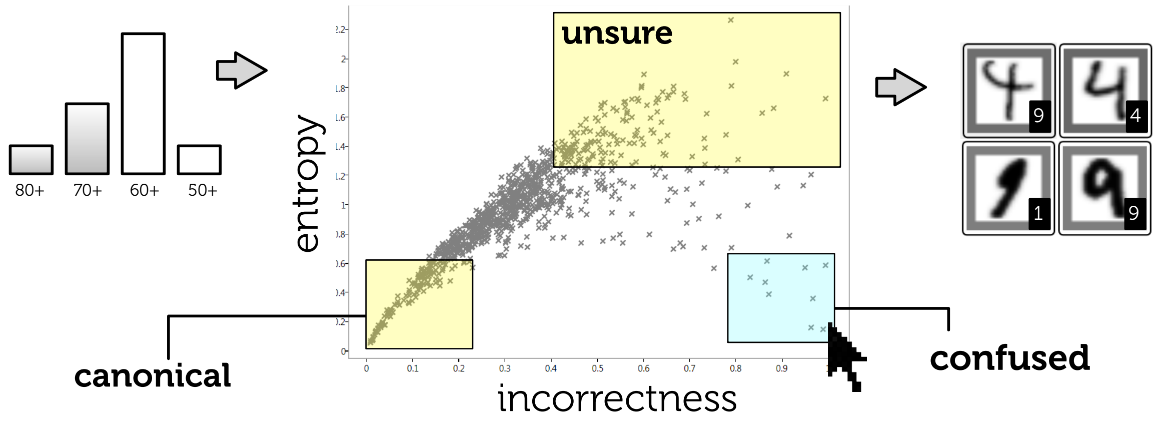


Figure 5.4: The middle image shows the incorrectness vs. entropy plot. To find label noise a programmer removes low accuracy configurations and inspects examples in the confused region.

Ground truth labels are often imperfect for a variety of reasons. For instance, humans are prone to make errors when fatigued, when distracted, or when presented with inherently ambiguous data. Noisy labels can adversely impact results, and they are often expensive to find because detecting them often requires human inspection. Prospect can use multiple models to reduce the number of examples a programmer inspects in debugging label noise.

Figure 5.4 (middle) shows a scatter plot visualization of two summary statistics for examples: incorrectness and label entropy. Incorrectness (x-axis) is the percentage of configurations that misclassify an example. Label entropy (y-axis) is the entropy of the distribution of labels predicted by each configuration for an example. Given the indicator function $1\{\}$, the set of configurations C, and the set of labels L; these values are computed as follows:

$$incorrectness(x_i) = \sum_{j}^{|C|} \frac{1\{y_i \neq \overline{y}_{ij)}\}}{|C|}$$

$$entropy(x_i) \quad = \quad -\sum_{l \in L} \frac{\sum_j^{|C|} 1\{\bar{y}_{ij} = l\}}{|C|} \times \ln \frac{\sum_j^{|C|} 1\{\bar{y}_{ij} = l\}}{|C|}$$

Figure 5.4 highlights three regions in the scatter plot. The canonical region contains examples that most configurations classify correctly (i.e., low-incorrectness, low-entropy). The unsure region contains examples for which different configurations generate widely varying predicted labels (i.e., high entropy). The confused region contains examples with high incorrectness and low entropy. These are the examples for which most configurations agree on a predicted label, but the predicted label is not the same as the ground truth label. These confused examples are of the most interest for detecting label noise, as the consistent misclassification by many different models suggests the ground truth label may be incorrect.

Figure 5.4 shows the exact steps a programmer would take within Prospect to find potential label noise. First, a programmer uses a configuration accuracy histogram to filter configurations with poor performance; this removes noisy models and reduces variance. Reducing high variance is important because it removes random noise from the system, more effectively separating the confused and canonical regions. Second, they select a set of relevant examples from the confused region of the scatter plot. Finally, they create a grid view to inspect those examples together with their ground truth labels. In this case, we can see that an example labeled nine is clearly a four and that an example labeled one may be a nine.

5.4 Generating New Features

Feature discovery is a complex process, with programmers often needing to acquire specialized domain knowledge before they can discover discriminative features. The key to

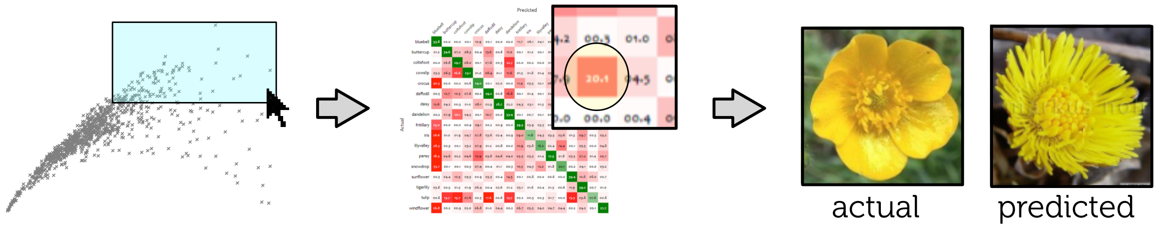


Figure 5.5: Looking at the confused examples in the unsure region can help programmers understand their data and create features. Here programmers are looking at a confusion in the aggregate confusion matrix.

discovering new discriminative features lies in understanding properties of the data that distinguish one class from another. Such understanding can be developed through deep analysis of the features or through analysis of how different examples are classified. Automated feature selection methods generally focus on analysis of the feature set, but this can be non-trivial for humans (especially in a high-dimensional feature space). On the other hand, looking at how different examples are classified is generally easier to comprehend and can provide insight into deficiencies of a feature set. Prospect provides visualizations to examine aggregate statistics regarding how different examples are classified and misclassified. This can in turn be used to identify situations where the current feature set is not sufficiently descriptive.

Figure 5.5 illustrates how Prospect supports the interactive process of finding examples to help build intuition for new features. The key idea is to enable programmers to observe a summarization of how different examples are classified, which should then help them develop insights about the differentiable characteristics of data. Figure 5.5's incorrectness vs. entropy plot again plays an important role in this process. To help develop insight into new potential features, I focus on the unsure region of the plot (examples with high entropy). These are misclassified by many configurations, but are not predominately mistaken for any particular class. These seem crucial to the process of feature discovery because their high entropy suggests that the available features cannot support reliable differentiation between classes. Focusing on the development of new features that are relevant to these examples should therefore provide new discriminative power to models.

Figure 5.5 shows the exact steps a programmer would take within Prospect to examine unsure examples to help develop insight for new features. A programmer first uses a rubberband tool to focus on a set of examples with high entropy and high incorrectness. After filtering to focus only upon these examples, they create an aggregate confusion matrix. This presents the normalized sum of confusion matrices for each configuration, highlighting which classes are commonly confused across all models. The programmer can then click into a cell in this aggregate confusion matrix to directly inspect examples that might benefit from new features. Prospect presents a representative example of a class together with each confused example. Here we can see that two examples have similar shape and color, but could likely be better differentiated by a new feature capturing image texture.

5.5 Experimental Setup

Both experimental studies were conducted on three multi-class datasets taken from different domains: the 20 newsgroups dataset [83], the MNIST digit recognition dataset [85], and the flowers image recognition dataset [104]. I use the same scheme for generating configurations in both evaluations. This section describes these common datasets and configurations.

5.5.1 Datasets

The newsgroups dataset consists of posts collected from 20 different public newsgroups. The original dataset consists of 20,000 documents across domains that are both related (e.g., `comp.sys.ibm.pc.hardware` and `comp.sys.mac.hardware`) and unrelated (e.g,. `misc.forsale` and `talk.politics.mideast`). I select 1,000 articles balanced by class and use the Weka API [137] to tokenize documents, stem words, and compute word count features (2057 features in total).

The digits dataset consists of 28x28 images of handwritten numerical digits from 0 to 9. The original dataset consists of 60,000 training and 10,000 test images which are normalized from their original dimensions and resized and normalized to fit into a 20x20 pixel box. I select 1,000 images balanced by class (100 for each digit) and use pixel values for features (784 features in total).

The flowers dataset consists of 1760 images of 17 types of flowers. Images vary in scale, pose, and lighting with large within-class variations. I select 680 examples balanced by class (40 per class). I use precomputed shape, color and texture features provided by Nilsback et al. [104].

5.5.2 Configurations

To create different configurations, I systematically vary the feature space, classification algorithm, training and testing split, and other parameters. Specifically, I create multiple feature spaces using feature selection based on information gain. I apply three different classification algorithms (support vector machines, decision trees, and Naïve Bayes) and vary parameters for each algorithm. I also randomly vary the composition of cross-validation folds. My implementation uses feature selection and classification algorithms provided by Weka.

I prune the set of configurations using two heuristics. First, in the interest of speed, I launch each configuration in a thread that terminates after one minute. If the execution does not terminate and produce a model I remove it from the list of configurations. Second, because I want a varied distribution of predicted labels, I remove configurations that apply the same classification algorithm and yield the same distribution of predicted labels.

My current process for automatically generating configurations is limited, as there is no guarantee the set of configurations adequately samples the space of possible configurations. Optimal sampling remains future work. Even with this current limitation, the results are promising.

5.6 Evaluating Label Noise Detection

The confused region can be seen as a classifier that attempts to identify examples that are incorrectly labeled. Because I want to minimize the examples a programmer inspects when identifying label noise, I want to maximize incorrectly labeled examples (true positives) while minimizing correctly labeled examples (false positives) in the confused region. I examine this tradeoff in comparison to two experimental baseline thresholds.

I first compare to sampling according to the posterior probability of the highest-accuracy model in the collection (sampling in order of increased confidence). This method finds examples about which the single best model has the least confidence. I choose this model for two reasons. First, programmers often debug using their highest-accuracy model. Second, good training performance is often indicative of low bias, and a model with low bias is likely to have a good true positive / false positive tradeoff. Success in this comparison thus shows the value of explicitly considering multiple models.

The second comparison is to a threshold considering only the incorrectness statistic. The confused region is differentiated from the unsure region according to the entropy of labels predicted by different configurations. This comparison confirms that both statistics provide distinct information to help in identifying noisy labels.

5.6.1 Procedure

To simulate label noise, I randomly selected 10 examples and randomly changed their ground truth labels. Consistent with the process illustrated in Figure 5.4, I removed poorly performing models by using only the top quartile of configurations (ordered by accuracy).

	incorrectness and entropy	incorrectness	single classifier
newsgroups	0.9978	0.9963	0.9929
digits	0.9952	0.9935	0.9878
flowers	0.9910	0.9804	0.9837

Table 5.1: For all three datasets the area under the ROC curve is higher if both incorrectness and entropy values are used to find label noise.

My baseline conditions classify examples based on threshold cutoffs. Examples below the threshold are correctly labeled, and examples above are incorrectly labeled. To compute ROC curves for the baselines, I set a high threshold value such that all examples are correctly labeled and then lower the threshold to compute the true positive and false positive rates.

I compare the baseline conditions to the confused region classifier. The confused region is a rectangle defined by two points. The bottom-right point is anchored, and the top-left point is variable. Examples within the rectangle are classified as incorrectly labeled. To compute the ROC curve for the confused region, I generate a set of possible confused regions by sampling different values for the top-left point and compute true-positive and false-positive rates for each possible confused region.

5.6.2 Results

Table 5.1 shows the area under the ROC curve for the three datasets. The area is high for all three conditions, but that is because ROC looks at the false positive and true positive rates. I have an unbalanced set of positive and negative examples, so the rates will not correspond to the actual number of false positives. Because I am trying to reduce the human cost of finding label noise, measured in the number of examples a human must verify, the number of false positives is important. Each new false positive is another example that must be manually verified.

To show the number of examples a human must inspect, I plotted the tradeoff between number of true positives (incorrectly labeled examples detected) and false positives (correctly labeled examples classified as mislabeled examples).

Figure 5.6 shows this tradeoff for the three datasets. The plots show that looking at the confused region (the solid red incorrectness and entropy line) results in fewer false positives for each true positive detected than the baseline conditions. Focusing on the confused region can reduce the number of examples a person must inspect by up to a factor of 3.

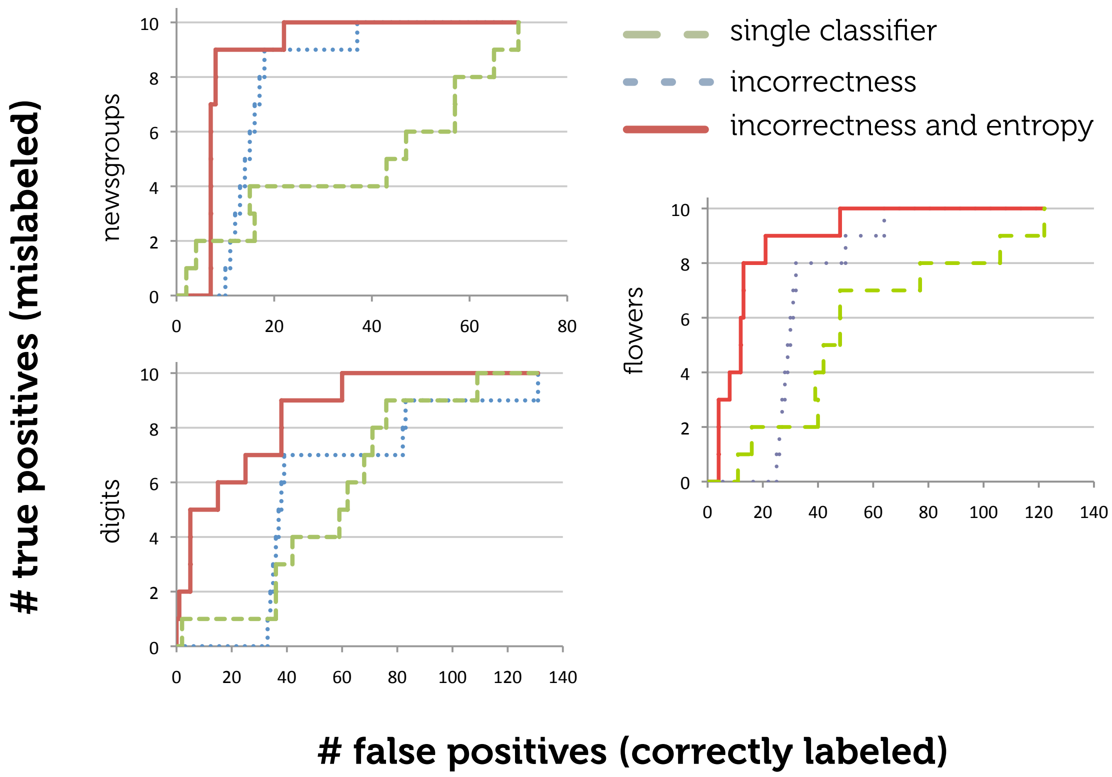


Figure 5.6: True positive vs. false positive for the three datasets. The incorrectness versus entropy line corresponds to the confused region. For all of the datasets, false positives occur inspecting examples within the confused region.

5.7 Evaluating Feature Generation

Human intuition is difficult to measure, and developing effective intuition about a feature often requires in-depth understanding of the problem. Instead of attempting to directly measure intuition, I assess how relevant examples are to the creation of new features. Relevant examples are those with the most capacity for improvement – those that will be "most helped" by the addition of new features. These should be most responsive to new features, exhibiting a larger change in correctness than other examples.

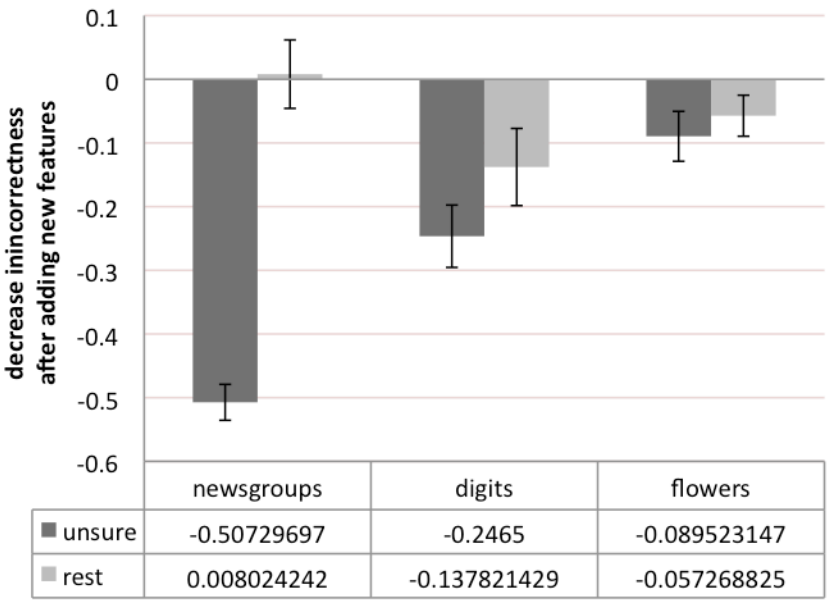
I claim that the unsure region contains relevant examples. To evaluate this claim, I conduct an experiment adding new descriptive features to a dataset and measuring the change in correctness for examples from the unsure region relative to the remainder of examples. We establish that examples from the unsure region are significantly more responsive to new descriptive features. This validates that programmer focus on the unsure region can be an effective strategy for developing insight to generate new features.

5.7.1 Procedure

The experiment requires two versions of each dataset: a *before* dataset and an *after* dataset (created by the addition of new features). I define the base datasets to be the after datasets, then create the before datasets by removing features. For the flowers dataset, I remove the shape feature. The newsgroup features are not as clearly divisible, so I remove a random 75% of the features. Random removal is ineffective in the digits data because adjacent pixel values provide redundant information. I therefore define the before features to be only the top-left region of each digit.

I define the unsure region in terms of the before dataset, thresholding at the top quartile of examples (according to the entropy of distributions from the before configurations). This splits the set of examples X into the unsure region U and the not unsure region $X - U$. I then compute δ_f, the change in correctness for each example resulting from adding additional features to create the after dataset. To test that U contains more relevant examples, I use a t-test to determine whether δ_f is significantly greater in U than in $X - U$.

It is possible that a greater δ_f in U is due to extreme entropy of the examples rather than relevance to a new feature (i.e., an effect similar to regression to the mean). To test this, we first generate a noise dataset by changing the random seed used to create cross-validation folds in the before dataset.



	newsgroups	digits	flowers
■ unsure	-0.50729697	-0.2465	-0.089523147
▨ rest	0.008024242	-0.137821429	-0.057268825

Figure 5.7: Incorrectness decreases after a new feature is added for examples within and outside of the unsure region. There is a significant difference for newsgroups and digits.

Changing the seed produces different training and validation sets, and different training sets will lead to changes in models, predicted labels, and entropy values. However, the underlying features are the same, only the training and verification splits differ. Since the features are the same, I refer to this difference as noise, and I use the noise value to determine if changes in entropy caused by the introduction of new features are significantly different than changes in entropy caused by the introduction of noise. I define δ_n as the difference due to noise, and use a t-test to determine whether δ_n is significantly greater in U than in $X - U$.

Because I find that examples in U are more responsive to noise, it is important to establish that the improvement to correctness that we see when adding relevant features δ_f is not explained by the noise δ_n. I examine this by testing that $\delta_f - \delta_n$ is significantly greater in U than in $X - U$. This validates the relevance of examples in the unsure region.

5.7.2 Results

Figure 5.7 shows a bar graph and table of the improvement in correctness resulting from the addition of new features. The correctness of unsure examples increases significantly more than other examples (newsgroups: $p < 0.01$, digits: $p < 0.01$, and flowers: $p < 0.04$). After accounting for variation due to noise, I observe that this effect is significant for the newsgroups ($p < 0.01$) and digits ($p < 0.01$) datasets. Although the improvement in correctness is not significant for the flowers dataset ($p \approx 0.27$), it is also not any worse. Consequently, I can conclude that focusing on the unsure examples can indeed provide programmers with insights critical to the process of interactive feature discovery.

In summary, the experiments indicate that focusing on examples in the unsure region can only help. The unsure region is a good place to start when trying to generate new features. While automatic discovery of new discriminative features is still a hard problem, Prospect can help a human programmer to explore the properties of the dataset and reason about such exogenous variables.

5.8 User Study

I conducted a user study in which participants provided me data they had collected and were trying to model. After processing their data with Prospect and creating raw data visualizations, I brought participants back to the lab and asked questions about examples in the unsure and confused region. I recruited three participants (P1-P3). To maintain anonymity, I refrain from describing their datasets in detail. P2 and P3 were working on multi-class problems, while P1 was working on a binary classification problem.

Automation Saves Time and Effort: P1 had just started analyzing and modeling, and Prospect allowed them to concentrate on features and data without worrying about tweaking parameters. P2 was in the middle of analyzing and refining their data using a single algorithm. Prospect helped them explore the space of configurations to find one that provided a significant accuracy boost (around 30%). P3 was near the end of their project and had painstakingly performed a manual exploration of configurations. They commented that Prospect would have reduced the effort of manual exploration. In all three cases Prospect either saved or would have saved participants time and effort.

Redesigning Data Collection: P1 found that their data was not rich enough to model their classes. Prospect helped them conclude that a redesign of their data collection mechanism would help create better models.

Finding Label Noise: P1 took steps to gather and verify clean data. P3 provided us an early version of their data with label noise, and the noise was easy to spot in the confused region. P2 had yet to perform a through data cleaning, and after reflection, they identified confused examples as label noise. Data cleaning can be problematic, so Prospect's ability to find label noise was useful.

Reinforcing Feature Ideas: P3 noted that the unsure region contained examples that were particularly hard to classify, and that they were modifying their approach to be robust to these types of errors. Reflecting on their data, the P3 and I were able to co-design a potential new feature. In fact, all three of the participants either came up with ideas for new features or reinforced ideas they already had. In this way, Prospect was able to guide the feature creation process.

5.9 Limitations

The current set of configurations, parameters, and pruning heuristics are hard coded. Although my studies show that these parameters work well for a variety of tasks, intelligently choosing which configurations to run might lead to the same results with less computation. Currently, Prospect removes redundant configurations after they have already been executed, but it may be possible to avoid redundancies online before Prospect runs a configuration. For example, each time Prospect needs to run a new configuration, it could look at what has already been run in the past and results from those configurations to inform the creation of a new configuration. For example, if Prospect sees that Naïve Bayes algorithms are not providing high quality models with varied results, it can use this information to choose a different classification algorithm when creating the next configuration.

When computing statistics, Prospect assumes that all configurations are equally important. Weighting configurations may lead to better results and provide programmers with the opportunity to steer results based on their preferences. By adding weights to components, Prospect can increase the weight for configurations that provide significantly different results. Additionally, if all configurations have weights, Prospect can run optimization algorithms to automatically learn weights based on different preferences (e.g., high accuracy, low redundancy).

The interactions that Prospect provides are based on a programmer exploring data by switching between visualizations and zooming in on examples. However, these interactions may be superfluous. My experiments show that certain types of examples are consistantly good to inspect when creating new features or finding mislabeled data. Instead of moving through visualizations, Prospect could provide an alternative interface based on questions. In such an interface, a programmer could simply click a button to indicate that they are looking for mislabeled examples. Based on this input, Prospect could automatically filter the dataset to only show examples in the confused region of the incorrectness and entropy plot.

5.10 Summary

Prospect provides a interactive visualization interface that leverages predictions from multiple models to help programmers understand their data. I have shown that Prospect works well for two common debugging tasks (detecting label noise and generating new features), and I believe there is potential to leverage multiple models to solve other questions about data (e.g., determining whether to collect more data).

Prospect automatically creates multiple models, but the creation of multiple models also happens organically as the result of experimentation. My studies show that supporting experimentation and the capturing the resulting models and predictions is important for supporting machine learning programming. In the next chapter, I discuss a new tool called Hindsight which is designed to help programmers as they experiment by changing data and code.

Chapter 6 | **HINDSIGHT**

The process of training a model involves iterative development of both data and code. During this iterative process, programmers may change their data, the features they extract from the data, the algorithm they use to model the data, their experimental methodology, and even the metrics by which they evaluate their model. As they make changes, they continually ask questions like: "Are my parameters tuned correctly? Is my data noisy? Do I need more expressive features? Do I need to collect more data? Have I chosen the right classification algorithm?"

But ultimately, programmers are just trying to answer one question: "Will this work?" More concretely: "Will my code and model be accurate enough or fast enough or predictable enough to solve my problem?" Often, this is not known *a priori*. When starting a classification project, the programmer is often not aware if it is even *possible* to solve the problem they are trying to solve with the data they are able to collect, with the features they can extract, and with the algorithms they have chosen. However, programmers may have a hypothesis that explains the reasons behind a poorly performing model. To find out if the behavior they want is possible, programmers run experiments to explore the space of possible functions.

Running experiments is an important part of using machine learning. Though experimentation, programmers begin to understand the details of the classification function a computer has learned. My studies show that programmers train better models when they are aware of the experimental process and make efforts to capture their own process. However, tracking experiments is tedious and hard. Often, programmers do not realize that they should track experiments at all until they find a systemic error that requires them to go back and revisit prior assumptions. In this chapter, I present Hindsight. Hindsight automates the process of capturing experimental history in order to help programmers reflect on what they have done and make better experimental decisions.

This chapter consists of three parts. First, I discuss the experimental process involved in training classifiers and the components of that process. Second, I discuss how Hindsight helps programmers structure their classification pipeline so that it can automatically track experimentation and how Hindsight supports the iterative process by making the stored history visible and usable. Finally, I discuss how the features provided in Hindsight can be extended to provide programmers with even better support for experimentation.

6.1 The Components of Experimentation

In this section, I discuss the key tasks involved in the process of experimentation with machine learning: creating and curating multiple alternatives, comparing results, and keeping an experimental history.

6.1.1 Multiple Alternatives

When a programmer asks, “Do I have enough data?,” they don’t necessarily care about the data in and of itself. They care about improving the performance of their model, and they hypothesize that an alternative model (i.e., one built with more data) will perform better. The exploration of multiple alternatives is a core part of programming with machine learning.

The programmer creates new alternatives in many different ways. Some alternatives are the product of an explicit hypothesis and an intentional change based on that hypothesis. In the example above, the programmer’s current understanding of the program’s behavior directs them to collect more data. Based on their understanding, they hypothesize that collecting more data will lead to better performance. The programmer expects a certain behavior.

Other alternatives are the byproduct of exploration and knowledge gathering based on an intuition about the way the model works. In these scenarios, the programmer doesn’t have any expectations about the model’s exact behavior, but does have an intuition about the meaning of the parameters used in the algorithm. For example, the depth of a decision tree determines how well the model fits the training data – deeper trees often produce better results. However, a tree that is too deep can overfit the training set and work poorly in practice. A programmer using decision trees would vary the depth of the tree and look at performance on the training and testing sets in order to determine which depth provides the most accurate classification. This process creates multiple different alternatives that vary along one parameter: the depth of the decision tree.

In contrast to the previous example, knowledge gathering can often be entirely undirected. For instance, a programmer might try using the same feature set with three different algorithms to see which one works better. In this scenario, the programmer doesn't have any intuition at all about how the algorithms work. They know that different algorithms can provide different results, and they are trying to cover their bases. As they continue to evolve their feature sets and datasets, they may periodically revisit their assumptions by re-exploring old algorithms or exploring new algorithms altogether.

6.1.2 Comparisons

Each alternative (which may be a change in data, a change in parameter, or a change in algorithm) has an associated result. This result is usually some sort of measurement. Recall that a cross-validation experiment provides predicted labels for all of the data in the dataset. Based on these predicted labels, programmers calculate summary statistics. The most commonly used summary statistic is accuracy. Accuracy provides an easy high-level metric for comparing two alternatives; the alternative with the higher accuracy is better.

Measurements are a key part of experimentation in programming with machine learning. The availability of concrete performance measurements separates machine learning from other exploratory programming tasks, such as designing an user interface. Measurements allow programmers to understand the effect of a change. For instance, a programmer comparing a Naïve Bayes algorithm to a support vector machine algorithm may find that the support vector machine is more accurate. Then, they will associate the change in accuracy with the change in algorithm. In this case, they will assume that support vector machines are better at modeling their data.

In practice, a single measurement is rarely sufficient. My studies show that programmers care about many different factors, some of which are easier to measure than others. Fundamentally, programmers make decisions by weighing tradeoffs. For example, a programmer using a Naïve Bayes algorithm might find that training and testing with this algorithm is much faster than with support vector machines and may decide that the increased accuracy is not worth the decreased training and testing speed. Tradeoffs are often very specific to the problem at hand, and programmers often need to revise their comparisons and measurements as they search through the space of possible functions.

6.1.3 Experimental History

One result of this search process is an experimental history consisting of changes to the program and the associated measurements and results for each change. In my studies, programmers who recorded their experimental history (either on paper or in file names) developed the best classifiers. Recording experimental history is also common practice in the natural sciences. Scientists keep lab notebooks with records of every experiment they try and the results of each experiment. These notebooks help scientists track their process so that they can review, communicate, and reproduce those experiments in the future.

Similarly, recording an experimental history allows a programmer to easily review what they have already tried. More broadly, reviewing past performance helps programmers make decisions about what to do in the future. For example, programmers may explore alternative classification algorithms when they make significant changes to their dataset or their feature generation code. Each alternative takes time to run. As the number of features increases and the number of examples increases, programmers may find that they can't test all of the desired alternatives in a reasonable amount of time. Reviewing the experimental history can help programmers prune their set of alternatives. For example, if a programmer finds that decision trees always perform poorly, they can remove them from future explorations.

Histories also help programmers rerun experiments after discovering a systemic error in their process. One example of a systemic error is when a programmer finds that their earlier evaluation metrics were flawed. Consider the scenario where a programmer finds that they have run the wrong type of cross-validation test for their chosen data set. In my studies, these errors were common. Programmers frequently ran random cross-validation tests when they should have run "leave-one-out" cross-validation tests. When programmers correct those errors, not only do they find that their current model works poorly, but they also find that they can't trust their results from prior experiments. Earlier decisions to settle on a specific algorithm or a specific feature set are called into question, and programmers have to revisit these decisions. Poor experimental tracking can make this process significantly more difficult. Without good tracking, it might be impossible to reproduce the conditions that led to certain results, requiring the programmer to start over again from the beginning.

Hindsight reduces the effort needed to keep track of experiments performed during the development process. It automatically tracks experimentation and stores structured histories. These histories help Hindsight support comparisons and revisiting history. In the next section, I discuss how Hindsight supports the components of experimentation.

6.2 Supporting Experimentation with Hindsight

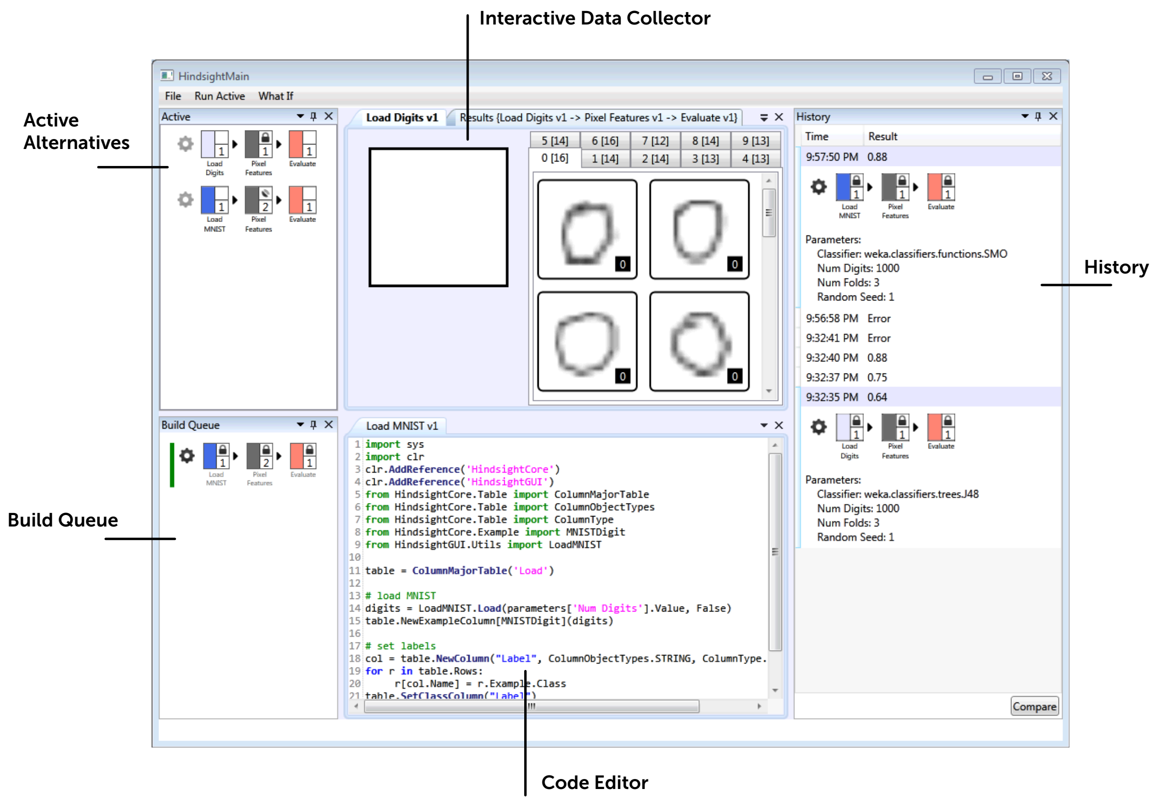
Hindsight (Figure 6.1) is a general-purpose machine learning programming environment that supports the development process around classification. Like Gestalt, Hindsight is not constrained to any single domain. It provides programmers with a structured representation of the classification pipeline, which helps them write well-organized code that defines the logic of a particular pipeline. Hindsight uses this structure to represent multiple alternatives, compare alternatives, capture an experimental history, and reuse histories to help programmers experiment more effectively.

6.2.1 Multiple Alternatives

Traditional programming environments (e.g., Visual Studio, Eclipse) focus on the development of a single program. In contrast, Hindsight is different from traditional programming environments in that it supports the concurrent evolution of multiple alternative programs as a high-level task. The curation and evolution of multiple potential solutions is key facet of the machine learning process. In Hindsight, the program represents an alternative classification pipeline. Programmers interact with alternative classification pipelines though the "active alternatives" panel (Figure 6.1 top left).

In Hindsight, a pipeline is called a *configuration*. Figure 6.2 shows two alternative configurations. The configuration consists of a list of *components* and their associated parameters. Each component has an input and output type. Like most data flow languages, two components can be connected together if the input type of the second component matches the output type of the first component. Fundamentally, a configuration is a data flow that describes the logic of a classification pipeline.

Programmers can define the logic of each component through code, or they can use interactive components that don't require explicit programming. The middle section of Figure 6.1 shows two different components: Load Data (top) and Load MNIST (bottom). While both components load digits, the way they load digit data differs. Load Data provides



```
import sys
import clr
clr.AddReference('HindsightCore')
clr.AddReference('HindsightGUI')
from HindsightCore.Table import ColumnMajorTable
from HindsightCore.Table import ColumnObjectTypes
from HindsightCore.Table import ColumnType
from HindsightCore.Example import MNISTDigit
from HindsightGUI.Utils import LoadMNIST

table = ColumnMajorTable('Load')

# load MNIST
digits = LoadMNIST.Load(parameters['Num Digits'].Value, False)
table.NewExampleColumn[MNISTDigit](digits)

# set labels
col = table.NewColumn("Label", ColumnObjectTypes.STRING, ColumnType.
for r in table.Rows:
     r[col.Name] = r.Example.Class
table.SetClassColumn("Label")
```

History

Time	Result
9:57:50 PM	0.88
9:56:58 PM	Error
9:32:41 PM	Error
9:32:40 PM	0.88
9:32:37 PM	0.75
9:32:35 PM	0.64

9:57:50 PM 0.88 — Load MNIST, Pixel Features, Evaluate

Parameters:
Classifier: weka.classifiers.functions.SMO
Num Digits: 1000
Num Folds: 3
Random Seed: 1

9:32:35 PM 0.64 — Load Digits, Pixel Features, Evaluate

Parameters:
Classifier: weka.classifiers.trees.J48
Num Digits: 1000
Num Folds: 3
Random Seed: 1

Figure 6.1: Hindsight supports the creation, curation, and execution of multiple alternative configurations of a classification pipeline. Programmers can construct configurations by combining components. Depending on the component, the logic of the component can be specified by writing code or using a graphical user interface. As programmers explore different alternatives, Hindsight records a history and presents an overview to programmers.

an interactive data collection interface where programmers can sketch new images. Programmers might build such an interface to provide a continuous stream of fresh data to their program. Load MNIST loads digit images from an existing data set using code. Programmers may need to write code when bootstrapping their system from available data sources. In Hindsight, these components are interchangeable. If a programmer has evolved a pipeline using their own data, they can easily switch sources to try the other dataset.

Configurations also have parameters. Parameters are a way for a programmer to specify commonly changing values without explicitly changing the code. For instance, a programmer might want to change the depth value of a decision tree algorithm or the number of folds in the cross-validation evaluation. In these scenarios, the change is too

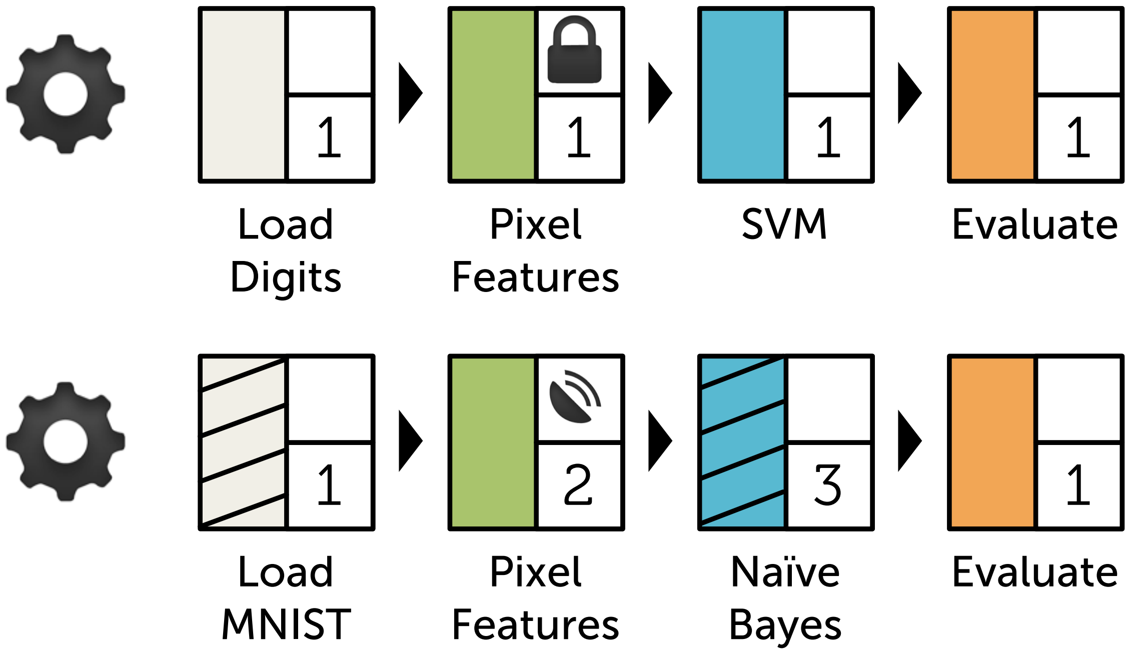
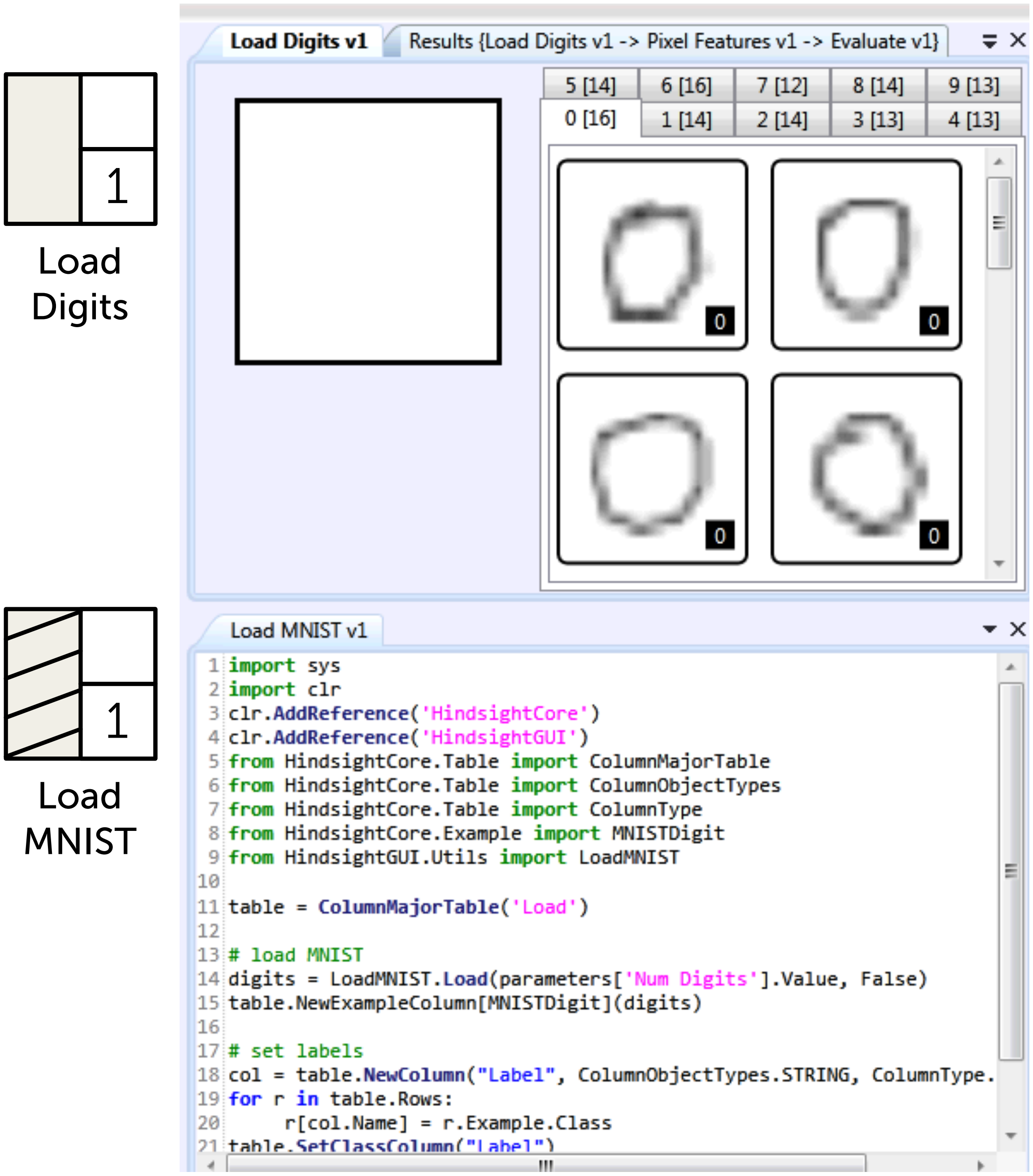


Figure 6.2: Two alternative configurations in Hindsight. The configurations are different, but do share some common components (i.e., Pixel Features and Evaluation).

small to justify creating a new version of the component because it does not constitute a significant semantic change to the behavior of the program.

In traditional programming environments, programmers have to add additional layers of computation to try multiple alternatives. For example, if they want to try multiple data sources, they have to write a "for" loop to iterate over those data sources. But in the traditional development environment, they still only see one program. The multiple alternatives are hidden by the opacity of the code. Hindsight surfaces these alternatives. Programmers can easily see all of the configurations they are working on, which helps them clearly understand the comparisons they are making.

In Hindsight, each component is versioned. When a programmer changes the logic of any component, Hindsight automatically updates the version. For example, the programmer changes the logic of the Load MNIST component in Figure 6.3 by writing code, and changes the logic of the component in Figure 6.3 by sketching new data. When the programmer runs the changed configuration, Hindsight saves the current updates and increments the versions of each component. Programmers can fork a component at any previous version in order to create a new component with the same logic. This allows programmers to take an existing



```
import sys
import clr
clr.AddReference('HindsightCore')
clr.AddReference('HindsightGUI')
from HindsightCore.Table import ColumnMajorTable
from HindsightCore.Table import ColumnObjectTypes
from HindsightCore.Table import ColumnType
from HindsightCore.Example import MNISTDigit
from HindsightGUI.Utils import LoadMNIST

table = ColumnMajorTable('Load')

# load MNIST
digits = LoadMNIST.Load(parameters['Num Digits'].Value, False)
table.NewExampleColumn[MNISTDigit](digits)

# set labels
col = table.NewColumn("Label", ColumnObjectTypes.STRING, ColumnType.
for r in table.Rows:
     r[col.Name] = r.Example.Class
table.SetClassColumn("Label")
```

Figure 6.3: Shows two alternative components with the same input and output types. In Hindsight, programmers can mix components are purely code with those that are more interactive. Load Digits provides an interactive sketching interface for gathering digit data, while load MNIST loads data from a file.

component and try two different paths for achieving the same behavior. A programmer might fork a component if they are trying to evolve and compare two different implementations of the same algorithm.

Configurations may have a combination of *shared* components and *unique components* (Figure 6.2). To make causal statements, programmers need to be able to compare configurations to see what has changed. Programmers can distinguish components by name, color, and version number. Color is an effective distinguishing factor because it allows programmers to quickly detect differences by glancing at their list of configurations. For example, it's easy to see that the configurations in Figure 6.2 have different data sources.

Each component also has a status. An updated symbol on the status box indicates that the component has changed since that last time the configuration was run. If any of the components in a configuration have changed, the programmer can re-run the configuration to get new results. In Hindsight, shared components are linked [95, 134]. This means that if programmers edit the code for the Experiment component in Figure 6.2, both configurations will be updated. Programmers can lock components to keep them from being updated. This allows them to hold aspects of the configuration static when making comparisons.

Over time, programmers will add and delete configurations as they learn more about their problem. For example, if a programmer discovers that decision trees are supposed to work well for their problem, they may want to create an alternative configuration that uses decision trees. Programmers can manually add new configurations by connecting components. They can also add new configurations when they fork a component or replace a component in an existing configuration. In this case, both the original and the new configuration are kept in the list of active configurations. Programmers can remove configurations by right-clicking on a component and selecting *delete* from the context menu.

6.2.2 Tracking History and Comparing Results

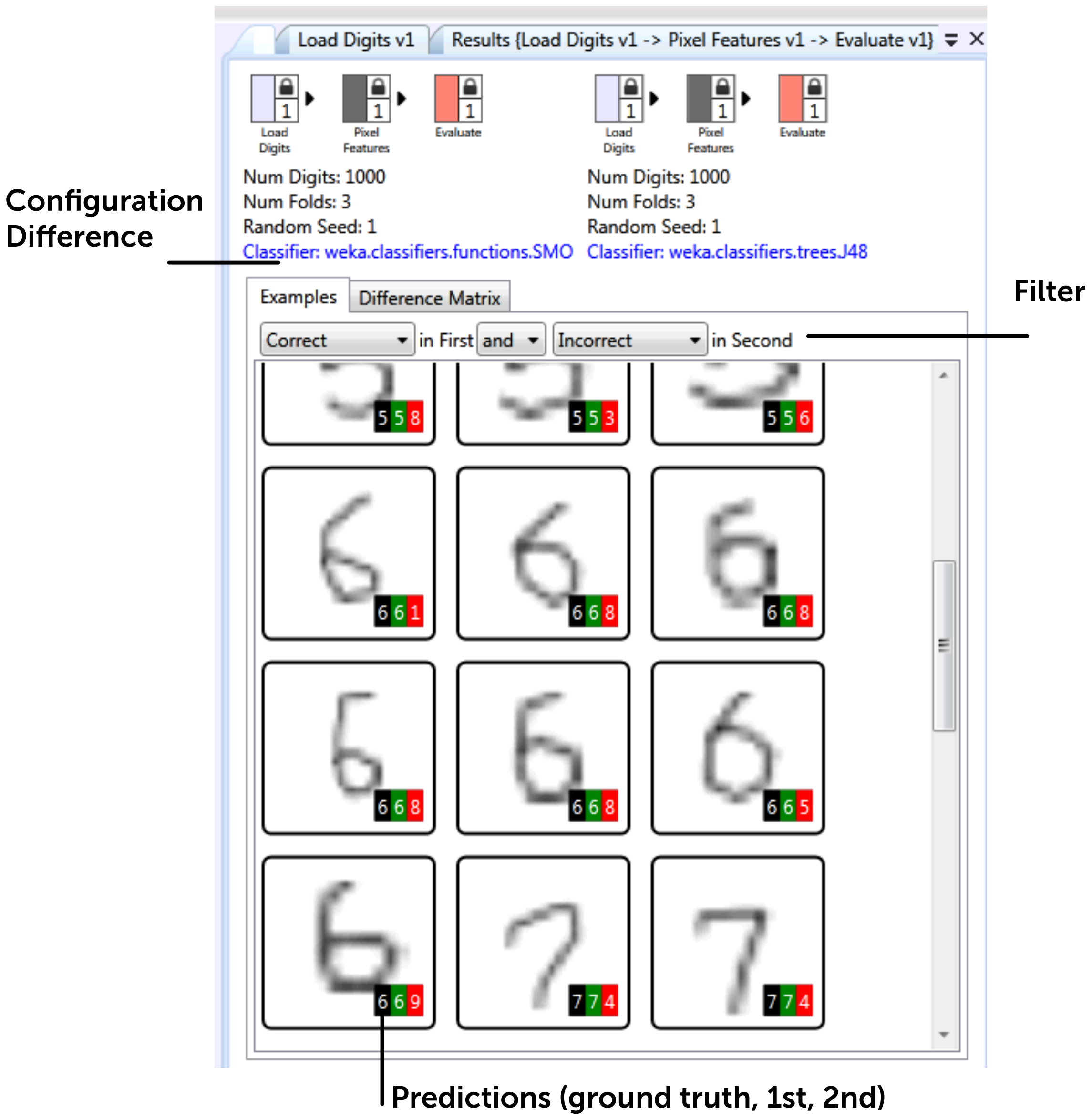
Hindsight automatically keeps a history of executed configurations and their results. Programmers can interact with this history using the history panel shown in Figure 6.1 right. The history panel provides a list of timestamps and accuracy measurements, which allows a programmer to see their progress over time. Programmers can expand each entry to see the specific configuration that resulted in each accuracy measurement, and they can double-click on the entry to see more information about the results.

Programmers need to see the differences between configurations in order to tie changes in code and data to changes in results. By having a structured representation of the pipeline and by capturing history, Hindsight can help programmers make useful comparisons. The history view itself provides a high-level comparison view. In this view, programmers can look at each configuration and its associated accuracy.

The explicit structure of a configuration is important for tracking history. Traditionally, programmers would have to manually track changes on paper and associate each change with an accuracy measurement (Chapter 3). My studies show that manual tracking for machine learning can be tedious and incomplete. Hindsight addresses both of these problems by automatically versioning each component every time a programmer runs a configuration. And, because Hindsight has access to all of the changes in program and the associated results, Hindsight captures the complete set of information needed to reproduce a particular experiment.

Programmers can make deeper comparisons by selecting two configurations from the history and opening the *comparison* view (Figure 6.4). The comparison view shows both configurations and highlights differences between them. The highlights allow programmers to understand the exact differences between implementations. Figure 6.4 top shows that the two configurations mostly agree, except for one parameter: the classification algorithm (in blue). The comparison view also visualizes the examples in the dataset so that programmers can dig deeper into the data behind the accuracy measurement. In this view, they can look at the areas of agreement and disagreement between individual classifiers. The bottom of Figure 6.4 shows a grid view of examples in the dataset. The visualization of each example contains the ground-truth label, the predicted label from the first configuration, and the predicted label from the section configuration. Programmers can see where the configurations agree and disagree by applying filters.

The comparison interface can also help programmers avoid common mistakes. For example, incomplete experiment logs can lead to errors in comparison. Consider the common scenario in which a programmer changes both the dataset and the classification algorithm at the same time and ends up with a lower accuracy. The programmer may forget that they have made changes to the dataset and improperly attribute the lower accuracy to the changed algorithm. Hindsight helps programmers avoid these mistakes. First, it explicitly shows differences between configurations so programmers can see exactly what they have



Num Digits: 1000
Num Folds: 3
Random Seed: 1
Classifier: weka.classifiers.functions.SMO

Num Digits: 1000
Num Folds: 3
Random Seed: 1
Classifier: weka.classifiers.trees.J48

Correct in First and Incorrect in Second

Figure 6.4: Hindsight allows programmers to look at the differences between both configurations and results. In this figure, Hindsight shows the examples that were correctly classified in the first configuration and incorrectly classified in the second configuration. The ground truth value, predicted value for the first configuration, and predicted value for the second configuration are shown on top of each example.

changed. Second, Hindsight can also detect *invalid* comparisons. For example, if two different datasets are being used, there will be zero overlap in the examples and Hindsight will notify the programmer of this fact.

Hindsight also provides a new visualization of the confusion matrix. In the comparison view (Figure **??**), programmers can click on the *diff* tab to look at the *diff confusion matrix* (Figure 6.5). The diff confusion matrix is an aggregate view of how classification results have changed between configurations. To build the diff confusion matrix, Hindsight first takes all of the overlapping examples between the two datasets and creates a confusion matrix for each of the configurations. Taking the diff is more than just subtracting one number from another, because the cells in the configuration confusion matrices may have the same value, but different examples. The diff confusion matrix has two values for each cell. The top value shows how many examples in the first configuration were not present in the second configuration and the bottom value shows how many examples in the second configuration were not present in the first configuration. Since Hindsight looks at the examples rather than just the values of the confusion matrix, it shows example-level changes that are normally hidden when comparing aggregates.

6.2.3 Putting History to Work

Hindsight uses the structured representation of the pipeline and the recorded history to help programmers overcome common experimental obstacles. Consider the scenario where a programmer is building a handwritten digit recognizer. A month into experimentation, they find that their evaluation metric is faulty. They were testing their algorithm using a random cross-validation, and since they are working with people, they should have been using a leave-one-out cross-validation technique. After correcting this mistake, changing their evaluation technique reduces the performance of their algorithm.

This is a systematic error in their experimental process. The problem here is not that the algorithm doesn't work as well as it should. The real problem can be found in the thought process behind the experiments. Each decision that they made (e.g. to include or exclude a feature, or to prefer one classification algorithm over another) was based on results that came from the incorrect evaluation metric. The correct thing to do would be to rerun the previous experiments to see if the experimental path they took was a good one. However, this is difficult in current programming tools. The tools do not maintain a log of what has been tried, and machine learning source code is often not structured so that components

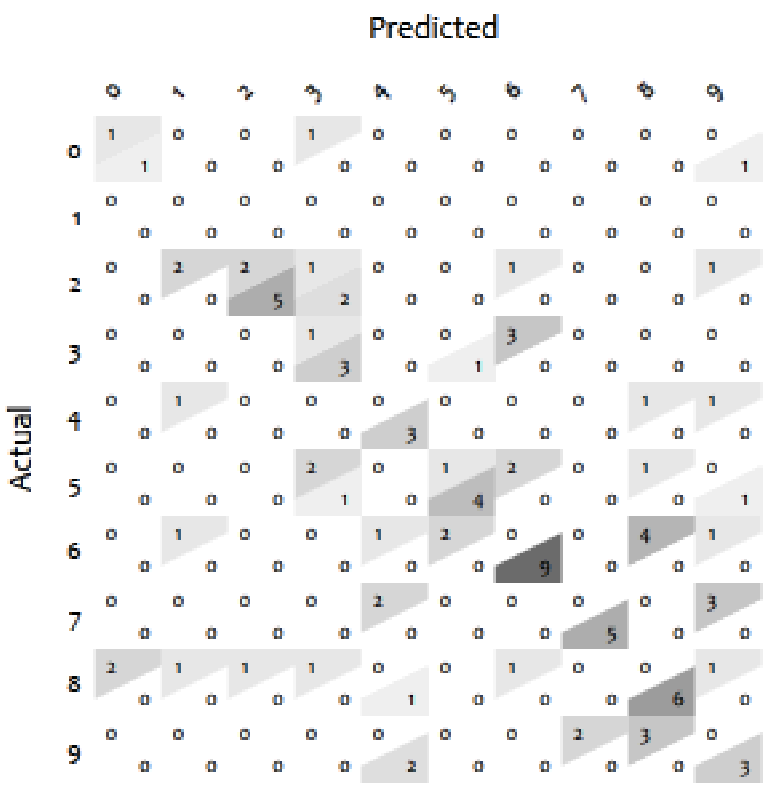
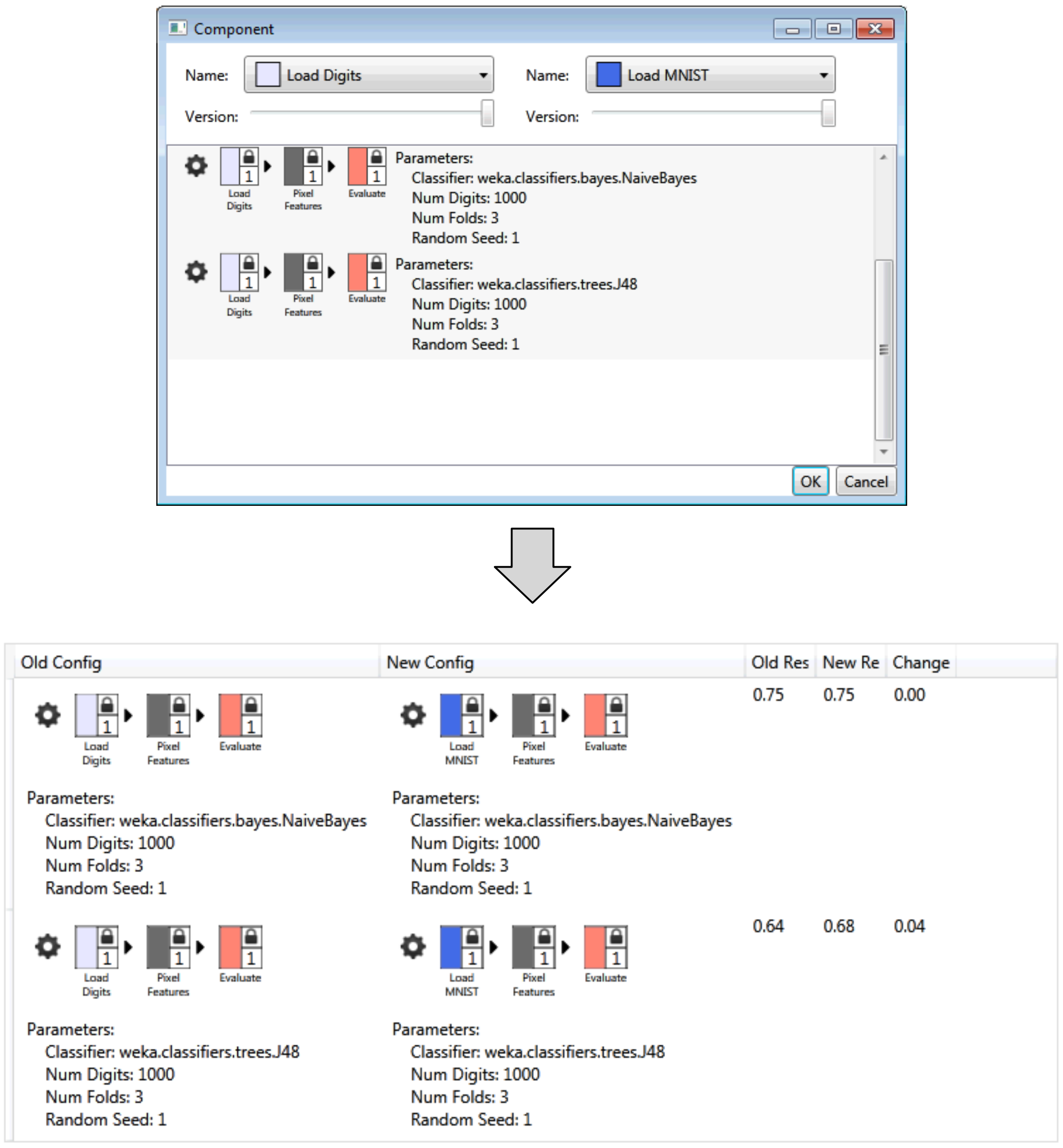


Figure 6.5: The diff confusion matrix shows the number of examples added to and removed from a confusion matrix cell between configurations. This matrix provides more information than just looking at the values of each confusion matrix. For example, the top left cell (zeros that are classified as zeros) shows that one example has been added by the second configuration and one has been removed by the second configuration. The value for the confusion matrix would be the same, but the specific examples within the cells are different.

can easily be swapped out.

Hindsight addresses this problem by allowing programmers to ask the following question directly in the tool: "What if I had used a leave-one-out cross-validation instead of a random cross-validation?" Programmers can pose this question by clicking on the "what-if" button. While posing the question, the programmer can choose which specific component to replace. In this case, they will choose to replace the evaluation component, and Hindsight will filter the history to include only the configurations that have the specified evaluation component. The programmer can then specify the new component they want to try and then click "run". Hindsight will rerun all of the previous experiments and present an overview of the new results. Results are sorted so that the experiments that have changed the most are at the top. This view allows programmers to see which experimental threads may be worth exploring again.

The ability to ask "what if?" is not just useful for recovering from systemic errors, where programmers know prior results are not trustworthy. Programmers often change components and then find that those decisions have an effect on prior experiments. For example, a programmer see how well their algorithms work with new data by changing the dataset (Figure 6.6), or they may find a new feature set and test the differences between their current implementation and the new implementation. In these cases, the programmer may need to revisit old assumptions. Having a structured approach allows Hindsight to capture history and make it easy to revisit old assumptions.



Name: Load Digits
Version:
Name: Load MNIST
Version:

Parameters:
Classifier: weka.classifiers.bayes.NaiveBayes
Num Digits: 1000
Num Folds: 3
Random Seed: 1

Parameters:
Classifier: weka.classifiers.trees.J48
Num Digits: 1000
Num Folds: 3
Random Seed: 1

Old Config	New Config	Old Res	New Re	Change
Parameters: Classifier: weka.classifiers.bayes.NaiveBayes; Num Digits: 1000; Num Folds: 3; Random Seed: 1	Parameters: Classifier: weka.classifiers.bayes.NaiveBayes; Num Digits: 1000; Num Folds: 3; Random Seed: 1	0.75	0.75	0.00
Parameters: Classifier: weka.classifiers.trees.J48; Num Digits: 1000; Num Folds: 3; Random Seed: 1	Parameters: Classifier: weka.classifiers.trees.J48; Num Digits: 1000; Num Folds: 3; Random Seed: 1	0.64	0.68	0.04

Figure 6.6: Hindsight reuses experimental histories to allow programmers to ask “what if?” questions about prior experiments. Here the programmer asks, “What if I had used Load MNIST as my dataset instead of Load Digits?” Hindsight finds all related configurations, replaces the component, reruns the experiments, and provides a comparison between old and new configurations.

6.3 Expanding Hindsight

Hindsight is a design exploration into how we can support experimentation in a programming environment. Currently, Hindsight provides some support for experimental tracking and the reuse of experimental history. This support helps automate some of the common tasks involved when running experiments. However, as Hindsight evolves, there will be new opportunities not only to better support common tasks, but also to leverage computation in order to augment the experimental process in new ways. In this section, I discuss some of these areas for expansion.

6.3.1 Automatic Hypothesis Testing

Explicitly formulating and discussing hypotheses is an important part of creating alternatives. As Hindsight evolves, new automated methods may ease the task of hypothesis formulation and management. One way of automating hypothesis testing is to represent some hypotheses as constraints on the system. For example, a programmer may run a few experiments evaluating algorithms and then choose support vector machines as their algorithm of choice. By choosing support vector machines, they are implicitly stating a belief. They are saying that for future datasets and feature sets, support vector machines will produce better models than other classification algorithms, such as decision trees or Naïve Bayes classifiers.

Hindsight could allow programmers to specify this belief as a constraint. The constraint would assert that for all future experiments, the accuracy of Naïve Bayes and decision trees should be lower than support vector machines. As the programmer continues to experiment, Hindsight would wait for free processor time and automatically run other algorithms in the background. If Hindsight ever finds that this constraint is broken, the tool would inform the programmer and suggest a new alternative path for experimentation.

6.3.2 Using Experimental Histories

As a programmer continues to experiment, the size of the experimental history will grow. An expanding experimental history provides opportunities to help guide future experimentation. After the programmer runs a series of experiments, Hindsight builds a dataset of experimental configurations and results. By analyzing patterns in this dataset, Hindsight can provide suggestions for new experiments that might yield good results. In the ideal case, Hindsight may even be able to run experiments automatically, evaluate if there are interesting differences, and present those differences to the programmer.

However, an expanding experimental history also introduces new challenges. As the number of experiments increases, understanding what has been tried and what has worked well might be difficult. A first step would be to provide an interactive query interface for programmers to filter and sort their experimental history. The interface could start as a command line tool in which programmers write SQL-like queries, but it could eventually evolve into a rich, interactive visualization tool. Building such a visualization system will require creating new visualization techniques that both summarize the performance of the system as a whole (e.g., a graph of accuracy over time) and provide ways to explore how configurations and individual components have changed over time (e.g., visualizing the evolution of the version control tree).

Another way to improve the organization of large experimental histories is to add new information. For example, the system can automatically tag configurations based on how they were created. Programmers could then filter out configurations that were created automatically or configurations that have results that are too similar to previous results. The system could also allow programmers to tag chosen configurations as important performance milestones and re-visualize the experimental history to focus on those configurations.

6.3.3 Multiple Comparisons

Hindsight currently supports the comparison of two alternatives. However, there are many cases where the programmer may need to compare the performance of many different alternatives at the same time. For example, a programmer might need to compare the performance of many different algorithms or look at the accuracy response of a single algorithm across a spectrum of parameter values. Hindsight can provide better support for these scenarios.

Instead of writing code to explore parameter values, programmers could directly interact with the visualizations to create and run new configurations. Consider the scenario where a programmer wants to see how the accuracy of a decision tree changes as max depth of the decision tree increases. The programmer could start by specifying the graph they want to see by selecting “depth” for the x-axis and “accuracy” for the y-axis. Then, they may specify the minimum and maximum values for the depth and the number of data points. Once specified, Hindsight would automatically run the specific configurations and populate the graph. The programmer could refine the graph by selecting a line segment and asking for more samples along that segment. Hindsight would then create and run more configurations to provide more data for that specific refinement.

6.4 Summary

When training the behavior of a model, programmers run experiments to search through the space of possible functions. They may change their data, the features they extract from the data, the algorithm they are trying to use to train a model, their experimental methodology, and even the metrics with which they evaluate their model. To support machine learning in general, a tool must support this specific experimental process.

In this chapter, I have presented Hindsight. Hindsight supports the creation and curation of multiple alternative classification pipelines. As programmers experiment, Hindsight automatically tracks changes to those pipelines and the results associated with those changes. It provides comparisons to help programmers evaluate alternatives and make informed decisions about next steps. By capturing the experimental process, Hindsight can leverage the history to help programmers easily ask “what if?” questions about their data.

In the next chapter, I discuss extensions to my current work and conclude my dissertation.

Chapter 7 | CONCLUSION

In the preceding chapters, I presented studies examining why training a model using machine learning is difficult. I then focused on classification algorithms and presented three tools that address these difficulties. In this chapter, I explore how these tools can be extended, and I conclude my dissertation.

7.1 Future Work

At the end of chapters 3 to 6, I described limitations of my existing work and opportunities for extending that work. These discussions were limited to the specific studies or tools in those chapters. In this section, I provide two directions for extending my research that cut across all of my tools and studies.

7.1.1 Data-Oriented Programming

Gestalt, Hindsight, and Prospect are currently focused on classification algorithms. Classification refers a specific set of machine learning teachniques. It assumes that programmers provide the learning algorithm with inputs and outputs (i.e., examples and labels). The algorithm trains a model that maps new inputs to outputs.

My tools and visualizations use the properties of classification to provide feedback to the programmer. For instance, all of my tools take advantage of the fact that examples have discrete ground truth and predicted labels. Gestalt provides confusion matrix visualizations that present a breakdown of predicted and ground truth labels. Prospect looks at agreement between predicted labels and ground truth labels to provide summary statistics. Hindsight's diff confusion matrix relies on the same information as a normal confusion matrix.

While some of the functionality will need to change, I believe the same interfaces can be extended to support other machine learning techniques. For example, Prospect could compare cluster similarity across different configurations. A modified version of Gestalt could provide filters based on continuous values for regression models.

There is also larger space of techniques that involve working with data and code to create models or programs. For example, many data-intensive computational tasks (e.g., visualization, gene alignment) also have a structured data flow pipeline [58] and can benefit from history tracking support [50]. Although Hindsight is currently configured for classification tasks, the underlying structure is general and can be extended to other domains. For example, Figure 7.1 shows Hindsight working for a gene alignment task.

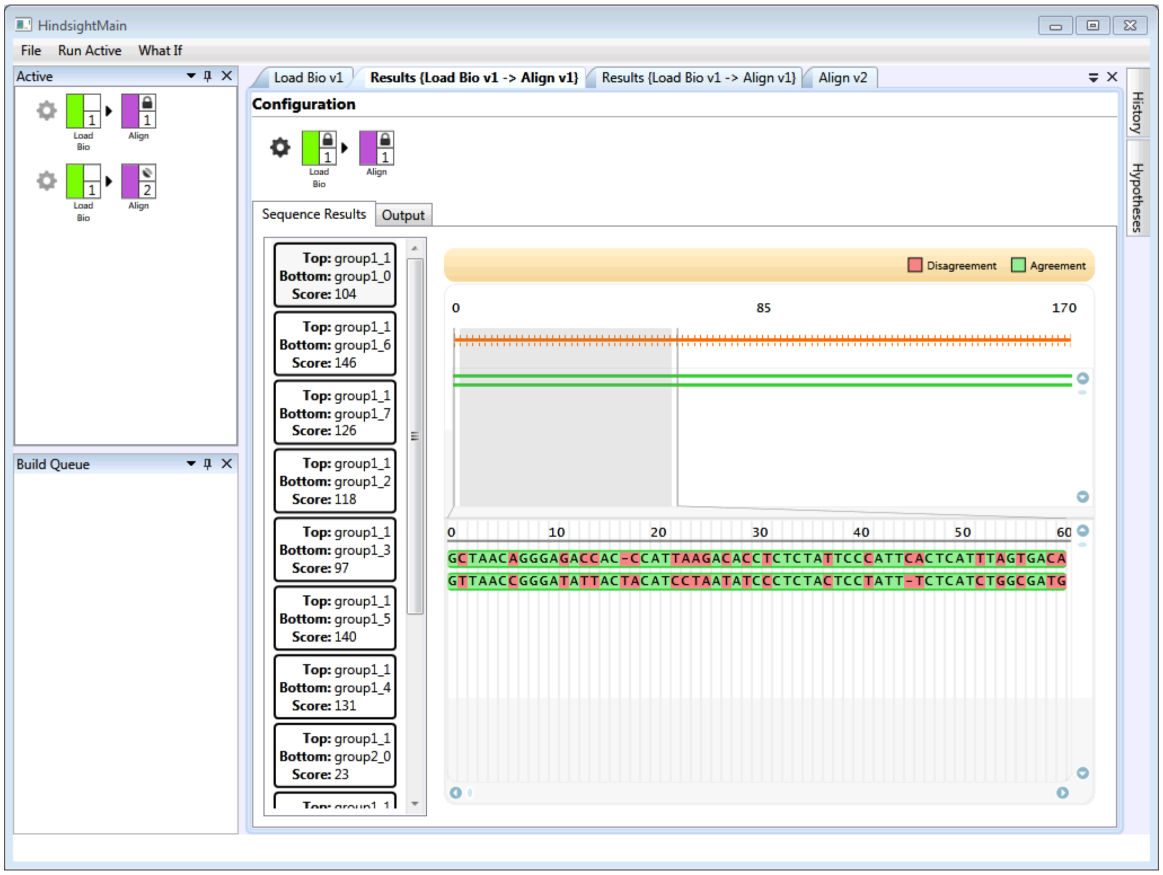
There is still a lot of work to be done before tools for training behavior are on par with tools for authoring code. However, working with data and creating models is necessary for tackling important problems. Discovering the common needs for these tasks and providing appropriate tools can help programmers build the software needed to solve these problems.

7.1.2 Big Data

My tools have been tested with relatively small datasets. These datasets are large enough that programmers cannot easily inspect all of the examples. They must create filters to reduce the dataset so they can inspect a manageable number of examples. However, the datasets are small enough for programmers to interactively train classifiers.

For many problems, the relevant data sets are large. For example, a recent image recognition classifier trained on YouTube videos looked at 10 million images [84]. The sheer difficulty of labeling such a dataset can mean that programmers have to use different techniques, each with their own draw backs. Unsupervised algorithms might find patterns, but programmers will have to make sense of groupings after the fact to understand what the system has learned. Automatic labeling based on external data can be error prone, and systemic errors can be hard to detect.

Additionally, large datasets take both a lot of time and computational power to model. Tools can be extended to provide partial results. Many algorithms provide intermediate models



```
Top: group1_1
Bottom: group1_0
Score: 104

Top: group1_1
Bottom: group1_6
Score: 146

Top: group1_1
Bottom: group1_7
Score: 126

Top: group1_1
Bottom: group1_2
Score: 118

Top: group1_1
Bottom: group1_3
Score: 97

Top: group1_1
Bottom: group1_5
Score: 140

Top: group1_1
Bottom: group1_4
Score: 131

Top: group1_1
Bottom: group2_0
Score: 23
```

```
0         10        20        30        40        50        60
GCTAACAGGGAGACCAC-CCATTAAGACACCTCTCTATTCCCATTCACTCATTTAGTGACA
GTTAACCGGGATATTACTACATCCTAATATCCCTCTACTCCTATT-TCTCATCTGGCGATG
```

Figure 7.1: Gene alignment programs have a similar pipeline structure to classification tasks: programmers need to be able to edit code and visualize data to iterate on their problem. The image above shows Hindsight extended to support the gene alignment problem. There are two working pipelines, the output of which can be inspected and compared (visualization taken from .NET Bio [8]).

that are not optimal but help programmers detect errors. A tool could also automatically pick small samples of a dataset to train smaller models quickly so programmers have some data to explore while their system is training a model on the larger dataset.

Even when the system provides results, understanding mistakes can be difficult. A 99% classification accuracy on a 10 million digit dataset means there are 100,000 incorrect examples the programmer must inspect. A tool could take errors and further decompose them by running clustering algorithms on the error set. Programmers could inspect smaller clusters to try to reason about errors.

7.2 Conclusion

This dissertation was about programming computers. It focused on a specific programming task: writing a function. A function takes inputs, performs operations, and provides outputs. Functions are a basic unit of programming. They are composed together to create more complex behaviors.

Typically, we program the behavior of a function by writing code. Programers manually specificy or author a function. However, there are important functions that can not be authored. We can still program some of these functions, but we must use different techniques. For example, we can program a function by training a computer using both code and data. The data provides examples of how we want the computer to behave, and the code tells the computer how to interpret those examples and model the data.

Training a model has a different process than authoring a function. I presented two studies that examined these differences for a specific set of training techniques, machine learning algorithms. These studies helped me better understand the process for training a model using machine learning, which I call the machine learning process. These studies also pointed at failures in current tool support for using machine learning.

The machine learning process involves an implementation task where a programmer creates a data flow pipeline using data and code. It also involves an analysis task where a programmer runs experiments and creates visualizations to understand the behavior of their data as it moves through the pipeline. Current machine learning tools are insufficient because no tool provides a good programming environment that supports the machine learning process. It is hard to move between implementation and analysis, focus on understanding data, and manage and track experiments.

To support the tasks of implementation and analysis, I created Gestalt, an integrated development environment designed around the machine learning process. Gestalt supports structuring a pipeline, writing code, and analyzing data. It provides interactive visualization capabilities to allow programmers to dig into errors to better understand the performance of the model. I showed that Gestalt's functionality helps programmers analyze the behavior of their model and effectively supports debugging an existing machine learning system.

With Gestalt, when programmers analyze data they are still looking at the result from one model. In this scenario, they may focus more on changing their model to improve performance rather than looking at properties of their data. To help programmers focus on understanding data, I created Prospect. Prospect was designed to help programmers spend more time thinking about data and less time thinking about code. Specifically, it generates multiple potential models for a single dataset by varying parameters, such as the machine learning algorithm and the feature set. It uses results from these multiple models to provide new insight about the dataset. My studies showed that Prospect can help programmers address important data centric problems such as noisy data and non-descriptive features.

This process of making changes and looking at results produces an experimentation history. To help programmers manage experiments and track history, I created Hindsight. Hindsight supports three main tasks in experimentation with machine learning: the creation of multiple alternatives, comparing results, and tracking history. By supporting these tasks and capturing an experimentation history, Hindsight can provide new functionality that takes advantage of historical data. For example, programmers can make a change to their data and apply that change to every single experiment they have run in the past.

These tools are three probes into the space of tool support for programming computers by training models. It is my hope that these tools provide inspiration for new tools that better understand the differences between authoring functions and training models. These new tools will go a long way to make training models as easy as authoring code is today, unleashing the creativity of developers and helping us solve problems in today's data rich world.

BIBLIOGRAPHY

[1] Abobe Creative Suite, `http://www.adobe.com/products/creativesuite.html`.

[2] Apple OSX, `http://www.apple.com/osx/`.

[3] dlwh/breeze, `https://github.com/dlwh/breeze`.

[4] Dropbox, `http://www.dropbox.com`.

[5] MATLAB, `http://www.mathworks.com/products/matlab/`.

[6] matplotlib: python plotting, `http://matplotlib.sourceforge.net/`.

[7] Microsoft Visual Studio, `http://www.microsoft.com/visualstudio/en-us`.

[8] .NET Bio, `http://bio.codeplex.com/`.

[9] Palantir, `http://www.palantir.com`.

[10] Simulink, `http://www.mathworks.com/products/simulink`.

[11] Spotfire, `http://spotfire.tibco.com/`.

[12] Tableau, `http://www.tableausoftware.com/`.

[13] Christopher Ahlberg, Christopher Williamson, and Ben Shneiderman. Dynamic queries for information exploration: An implementation and evaluation. *CHI*, 1992.

[14] Saleema Amershi, James Fogarty, Ashish Kapoor, and Desney Tan. Overview based example selection in end user interactive concept learning. *UIST*, 2009.

[15] Saleema Amershi, James Fogarty, Ashish Kapoor, and Desney Tan. Examining multiple potential models in end-user interactive concept learning. *CHI*, 2010.

[16] Saleema Amershi, James Fogarty, and Daniel S. Weld. ReGroup: interactive machine learning for on-demand group creation in social networks. *CHI*, 2012.

[17] Saleema Amershi, Bongshin Lee, Ashish Kapoor, Ratul Mahajan, and Blaine Christian. CueT: Human-Guided Fast and Accurate Network Alarm Triage. *CHI*, 2011.

[18] Mihael Ankerst, Christian Elsen, and Martin Ester. Visual classification: an interactive approach to decision tree construction. *KDD*, 1999.

[19] Larry Arnstein, Chia-Yang Hung, Robert Franza, Qing Hong Zhou, Gaetano Borriello, Sunny Consolvo, and Jing Su. Labscape: a smart environment for the cell biology laboratory. *IEEE Pervasive Computing*, 1(3), 2002.

[20] Daniel Ashbrook and Thad Starner. MAGIC: a motion gesture design tool. *CHI*, 2010.

[21] Sumit Basu, Steven M. Drucker, and Hao Lu. Assisting Users with Clustering Tasks by Combining Metric Learning and Classification. *AAAI*, 2010.

[22] Barry Becker, Ron Kohavi, and Dan Sommerfield. Visualizing the Simple Bayesian Classifier. In *Information Visualization in Data Mining and Knowledge Discovery*, chapter 18, pages 237–250. 2001.

[23] Richard A. Becker and William S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2), 2012.

[24] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. KNIME: The Konstanz Information Miner. *SIGKDD Explorations*, 11(1), 2008.

[25] Peter A. Blume. *The LabVIEW Style Book*. 2007.

[26] Michael Bostock and Jeffrey Heer. Protovis: a graphical toolkit for visualization. *VIS*, 2009.

[27] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3: Data-Driven Documents. *VIS*, 2011.

[28] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, Scott R. Klemmer, and San Francisco. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. *CHI*, 2009.

[29] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2), 1996.

[30] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. Managing the Evolution of Dataflows with VisTrails. *ICDEW*, Toxicological Centre, University of Antwerp, Universiteitsplein 1, 2610 Antwerp, Belgium., 2006.

[31] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. VisTrails: Visualization meets Data Management. *SIGMOD*, 2006.

[32] Doina Caragea, Dianne Cook, Hadley Wickham, and Vasant Honavar. Visual Methods for Examining SVM Classifiers. In *Visual Data Mining: Theory, Techniques, and Tools for Visual Analytics*, pages 136–153. 2008.

[33] Per Cederqvist. *Version Management with CVS*. 2002.

[34] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2007.

[35] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*, volume 1. 2008.

[36] P.T. Cox, F.R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. *VL*, 1989.

[37] Joseph A. Cruz and David S. Wishart. Applications of Machine Learning in Cancer Prediction and Prognosis. *Cancer Informatics*, 2, 2006.

[38] Janez Demšar, Blaz Zupan, Gregor Leban, and Tomaz Curk. Orange: From Experimental Machine Learning to Interactive Data Mining. *KDD*, 2004.

[39] Brenda Dervin. From the mind's eye of the user: The sense-making qualitative-quantitative methodology. In Jack D Glazier and Ronald R Powell, editors, *Qualitative research in information management*, Qualitative Research In Information Management, chapter 14, pages 61–84. 1992.

[40] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. a CAPpella: programming by demonstration of context-aware applications. *CHI*, 2004.

[41] Steven P. Dow, Julie Fortuna, Dan Schwartz, Beth Altringer, Daniel L. Schwartz, and Scott R. Klemmer. Prototyping Dynamics : Sharing Multiple Designs Improves Exploration, Group Rapport, and Results. *CHI*, 2011.

[42] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction*, 17(4), 2010.

[43] Steven M. Drucker, Danyel Fisher, Sumit Basu, and Microsoft Way. Helping Users Sort Faster with Adaptive Machine Learning Recommendations. *INTERACT*, 2011.

[44] Jacky Estublier, David Leblang, André Van Der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4), 2005.

[45] Jerry Alan Fails and Dan R. Olsen. A Design Tool for Camera-based Interaction. *CHI*, 2003.

[46] Jerry Alan Fails and Dan R. Olsen. Interactive Machine Learning. *IUI*, 2003.

[47] Rebecca Fiebrink, Dan Trueman, and Perry R. Cook. A Meta-Instrument for Interactive , On-the-fly Machine Learning. *NIME*, 2009.

[48] James Fogarty and Scott E. Hudson. Toolkit support for developing and deploying sensor-based statistical models of human situations. *CHI*, 2007.

[49] James Fogarty, Desney Tan, Ashish Kapoor, and Simon Winder. CueFlik: Interactive Concept Learning in Image Search. *CHI*, 2008.

[50] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: a survey. *Computing in Science and Engineering*, 10(3), 2008.

[51] James Frew, Dominic Metzger, and Peter Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice & Experience*, 20(5), 2008.

[52] Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. Technical report, 1998.

[53] Philip J. Guo and Margo Seltzer. Burrito: Wrapping Your Lab Notebook in Computational Infrastructure. *TaPP*, 2012.

[54] Björn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. *CHI*, 2007.

[55] Björn Hartmann, Loren Yu, Abel Allison, and Yeonsoo Yang. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. *CHI*, 2008.

[56] Jeffrey Heer and Maneesh Agrawala. Software design patterns for information visualization. *VIS*, 2006.

[57] Jeffrey Heer and Maneesh Agrawala. Design considerations for collaborative visual analytics. *VAST*, 2007.

[58] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. *CHI*, 2005.

[59] Jeffrey Heer, Fernanda B Viégas, and Martin Wattenberg. Voyagers and Voyeurs : Supporting Asynchronous Collaborative Information Visualization. *CHI*, 2007.

[60] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[61] Mark R. Hodges and Martha E. Pollack. An 'Object-Use Fingerprint': The Use of Electronic Sensors for Human Identification. *UbiComp*, 2007.

[62] Raphael Hoffmann, Saleema Amershi, Kayur Patel, Fei Wu, James Fogarty, and Daniel S. Weld. Amplifying community content creation with mixed initiative information

extraction. *CHI*, 2009.

[63] Eric Horvitz. Principles of mixed-initiative user interfaces. *CHI*, 1999.

[64] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. *PLDI*, 1990.

[65] J.W. Hunt and M.D. Mcilroy. An Algorithm for Differential File Comparison. Technical Report 41, Bell Labs, 1976.

[66] Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3), 1996.

[67] Jensen Harris. No distaste for paste, `http://blogs.msdn.com/b/jensenh/archive/2006/04/07/570798.aspx`, 2006.

[68] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1), 2004.

[69] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? *ICSE*, 2009.

[70] Ashish Kapoor, Bongshin Lee, Desney Tan, and Eric Horvitz. Interactive optimization for steering machine classification. *CHI*, 2010.

[71] Josef Kittler, Mohamad Hatef, Robert P.W. Duin, and Jiri Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3), 1998.

[72] Scott R. Klemmer, Mark W. Newman, Ryan Farrell, Mark Bilezikjian, and James A. Landay. The Designer's Outpost: A Tangible Interface for Collaborative Web Site Design. *UIST*, 2001.

[73] Scott R. Klemmer, Michael Thomsen, Ethan Phelps-Goodman, Robert Lee, and James A. Landay. Where Do Web Sites Come From? Capturing and Interacting with Design History. *CHI*, 2002.

[74] Donald E. Knuth. The Errors of TEX. *Software: Practice and Experience*, 19(7), 1989.

[75] Andrew J. Ko and Brad A. Myers. Development and evaluation of a model of programming errors. *VLHCC*, 2003.

[76] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. *CHI*, 2004.

[77] Andrew J. Ko and Brad a. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2), 2005.

[78] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. *ICSE*, 2008.

[79] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six Learning Barriers in End-User Programming Systems. *VLHCC*, 2004.

[80] Ron Kohavi and David H. Wolpert. Bias Plus Variance Decomposition for Zero-One Loss Functions. *ICML*, 1996.

[81] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3), 2006.

[82] Todd Kulesza, Weng-keen Wong, Simone Stumpf, Stephen Perona, Rachel White, Margaret M. Burnett, Ian Oberst, and Andrew J. Ko. Fixing the Program My Computer Learned: Barriers for End Users, Challenges for the Machine. *IUI*, 2009.

[83] Ken Lang. NewsWeeder: Learning to Filter Netnews. *ICML*, 1995.

[84] Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeff Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. *ICML*, 2012.

[85] Yann Lecun, Lèon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11), 1998.

[86] Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. 2009.

[87] Allan Christian Long, James A. Landay, and Lawrence A. Rowe. Implications for a gesture design tool. *CHI*, 1999.

[88] Ada Lovelace. Translator notes to "Sketch of the Analaytical Engine Invented by Charges Babbage, Esq.". *Scientific Memoirs*, 3, 1843.

[89] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18(10), 2006.

[90] Peter Macko, Margo Seltzer, and Kiran-Kumar Muniswamy-Reddy. Provenance for the cloud. *FAST*, 2010.

[91] Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. Eyepatch: Prototyping Camera-based Interaction through Examples. *UIST*, 2007.

[92] Donald Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136), 1968.

[93] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. *KDD*, 2006.

[94] Lance A. Miller and John C. Thomas. Behavioral Issues in the use of interactive systems. *International Journal of Man-Machine Studies*, 9(5), 1977.

[95] Robert C. Miller and Brad A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. *USENIX*, 2001.

[96] Tom M. Mitchell. *Machine Learning*. 1997.

[97] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. Layering in provenance systems. *USE*, 2009.

[98] Kiran-Kumar Muniswamy-Reddy and David a. Holland. Causality-based versioning. *ACM Transactions on Storage*, 5(4), 2009.

[99] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. *USENIX*, 2006.

[100] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7(1), 2000.

[101] Brad A. Myers and David S. Kosbie. Reusable Hierarchical Command Objects. *CHI*, 1996.

[102] Brad A. Myers, David A. Weitzman, Andrew J. Ko, and Duen Horng Chau. Answering why and why not questions in user interfaces. *CHI*, 2006.

[103] Mark W. Newman and James A. Landay. A Sketch of Web Site Design Practice. *DIS*, 2000.

[104] Maria-Elena Nilsback and Andrew Zisserman. A Visual Vocabulary for Flower Classification. *CVPR*, 2006.

[105] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice & Experience*, 18(10), 2006.

[106] Bryan O'Sullivan. *Mercurial: The Definitive Guide*, volume 7. 2009.

[107] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. *EMNLP*, 2002.

[108] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James A. Landay. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning ProcessesNo Title. *UIST*, 2010.

[109] Kayur Patel, Steven M. Drucker, James Fogarty, Ashish Kapoor, and Desney S. Tan. Using Multiple Models to Understand Data. *IJCAI*, 2008.

[110] Kayur Patel, James Fogarty, James A. Landay, and Beverly Harrison. Investigating Statistical Machine Learning as a Tool for Software Development. *CHI*, 2008.

[111] Alan Ritter and Sumit Basu. Learning to generalize for complex selection tasks. *IUI*, 2009.

[112] Nick Rizzolo and Dan Roth. Learning based java for rapid development of nlp systems. *LREC*, 2010.

[113] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structure, and Architecture*. 1996.

[114] Dean Rubine. Specifying gestures by example. *SIGGRAPH*, 1991.

[115] Daniel M. Russell, Mark J. Stefik, Peter Pirolli, and Stuart K. Card. The cost structure of sensemaking. *INTERACT*, 1993.

[116] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A Bayesian approach to filtering junk e-mail. *AAAI Workshop on Learning for Text Categorization*, 1998.

[117] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. *ACM SIGOPS Operating Systems Review*, 34(2), 2000.

[118] Robert E. Schapire. The boosting approach to machine learning: an overview. *MSRI Workshop on Nonlinear Estimation and Classification*, Shannon Laboratory, 2002.

[119] Carlos E. Scheidegger, Huy T. Vo, David Koop, and Juliana Freire. Querying and Re-Using Workflows with VisTrails. *SIGMOD*, 2008.

[120] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Cláudio T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6), 2007.

[121] Ben Shneiderman. Visual User Interfaces for Information Exploration. *ASIS&T*, 1991.

[122] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6), 1994.

[123] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. *VL*, 1996.

[124] Ben Shneiderman. Creating Creativity : User Interfaces for Supporting Innovation. *Transactions on Computer-Human Interaction*, 7(1), 2000.

[125] Rok Sosič and David Abramson. Guard: A relative debugger. *Software: Practice and Experience*, 27(2), 1997.

[126] Christopher Stolte, Diane Tang, and Pat Hanrahan. Polaris: A System for Query, Analysis, nd Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1), 2002.

[127] Simone Stumpf, Vidya Rajaram, Lida Li, Margaret Burnett, Thomas Dietterich, Erin Sullivan, Russell Drummond, and Jonathan Herlocker. Toward Harnessing User Feedback for Machine Learning. *IUI*, 2007.

[128] William Robert Sutherland. *The on-line graphical specification of computer procedures*. PhD thesis, MIT, 1963.

[129] Justin Talbot, Bongshin Lee, Ashish Kapoor, and Desney S. Tan. EnsembleMatrix: interactive visualization to support machine learning with multiple classifiers. *CHI*, 2009.

[130] Michael Terry and Elizabeth D. Mynatt. Recognizing creative needs in user interface design. *C&C*, 2002.

[131] Michael Terry and Elizabeth D. Mynatt. Side Views : Persistent , On-Demand Previews for Open-Ended Tasks. *UIST*, 2002.

[132] Walter F. Tichy. Rcs — a system for version control. *Software: Practice and Experience*, 15(7), 1985.

[133] Ryan Tomayko. Introducing GitHub Compare View, `https://github.com/blog/612-introducing-github-compare-view`, 2010.

[134] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing Duplicated Code with Linked Editing. *VLHCC*, 2004.

[135] Daniel S. Weld, Raphael Hoffmann, and Fei Wu. Using Wikipedia to bootstrap open information extraction. *ACM SIGMOD Record*, 37(4), 2009.

[136] Christopher Williamson and Ben Shneiderman. The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real-Estate Information Exploration. *SIGIR*, 1992.

[137] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd edition, 2005.

[138] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *UIST*, 2007.

[139] Kang Zhang. *Visual Languages And Applications*. 2007.

[140] George Kingsley Zipf. *Human behavior and the principle of least effort*. 1949.