

Gestalt: Integrated Support for Implementation and Analysis in Machine Learning

Kayur Patel[†], Naomi Bancroft[†], Steven M. Drucker[‡], James Fogarty[†], Andrew J. Ko⁺, James A. Landay[†]

[†]Computer Science & Engineering

DUB Group, University of Washington
Seattle, WA 98195

{ kayur, bancron, jfogarty, landay }@cs.washington.edu

[‡]Microsoft Research

One Microsoft Way
Redmond, WA 98052

sdrucker@microsoft.com

⁺The Information School

DUB Group, University of Washington
Seattle, WA 98195

ajko@u.washington.edu

ABSTRACT

We present Gestalt, a development environment designed to support the process of applying machine learning. While traditional programming environments focus on source code, we explicitly support both code and data. Gestalt allows developers to *implement* a classification pipeline, *analyze* data as it moves through that pipeline, and *easily transition* between implementation and analysis. An experiment shows this significantly improves the ability of developers to find and fix bugs in machine learning systems. Our discussion of Gestalt and our experimental observations provide new insight into general-purpose support for the machine learning process.

Author Keywords

Gestalt, machine learning, software development.

ACM Classification Keywords

H5.2 Information Interfaces and Presentation: User Interfaces;
D2.6 Programming Environments: Integrated Environments.

INTRODUCTION AND MOTIVATION

Machine learning is at the core of many advances in science and technology. Within HCI, researchers have applied machine learning to search [9], facilitating creativity [18], and helping people live healthier lives [6]. Within computer science, machine learning can reduce system downtime [3] and detect anomalous network behavior [5]. In humanity's greatest pursuits, machine learning can help understand cancer [7] and the beginnings of the universe [1].

Despite the sophistication of machine learning methods and their widespread impact in research, these algorithms are seldom applied in practice by ordinary software engineers. One reason is that applying machine learning is difficult in ways *different* than traditional programming. Traditional programming is often discrete and deterministic, but most machine learning is stochastic. Traditional programming focuses on modules and lines of code, but machine learning focuses on pipelines and data. Traditional programming is

often debugged with print statements and breakpoints, but machine learning requires analyses with visualizations and statistics. Traditional programming allows developers to explicitly *describe* the behavior of a program, but systems that use machine learning must *learn* behavior from data. Developers need new methods and tools to support the task of applying machine learning to their everyday problems.

Prior research has examined domain-specific support for applying machine learning to solve several important problems. Crayons uses a coloring metaphor for training image segmentation classifiers [8]. Eyepatch allows composition and training of classifiers to create vision systems. Exemplar supports direct manipulation methods for specifying simple sensor-based recognizers [14]. 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 developer to create a system. But these same limitations also constrain the developer if a tool's assumptions do not match the developer's needs.

This paper presents Gestalt, a general-purpose tool for applying machine learning. Gestalt targets developers, providing full support for writing code to specify the series of steps in a classification pipeline (Figure 1). In supporting a wide range of classification problems, Gestalt generalizes the lessons of prior domain-specific tools. Specifically, Gestalt allows developers to *implement* a classification pipeline, *analyze* data as it moves through that pipeline, and *easily transition* between implementation and analysis.

The specific contributions of this work include:

- Discussion of general-purpose development environment support for the application of machine learning.
- The Gestalt development environment. Gestalt supports the *implementation* of a classification pipeline, *analysis* of data as it moves through that pipeline, and *easy transitions* between implementation and analysis.
- Discussion of Gestalt's capabilities, including a focus on generalizing lessons from domain-specific tools to provide general-purpose support for machine learning.
- An evaluation demonstrating that Gestalt significantly improves developer ability to find and fix bugs in two typical applications of machine learning.
- Discussion of current limitations and future opportunities for general-purpose machine learning support.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

UIST'10, October 3–6, 2010, New York, New York, USA.

Copyright 2010 ACM 978-1-4503-0271-5/10/10....\$10.00.

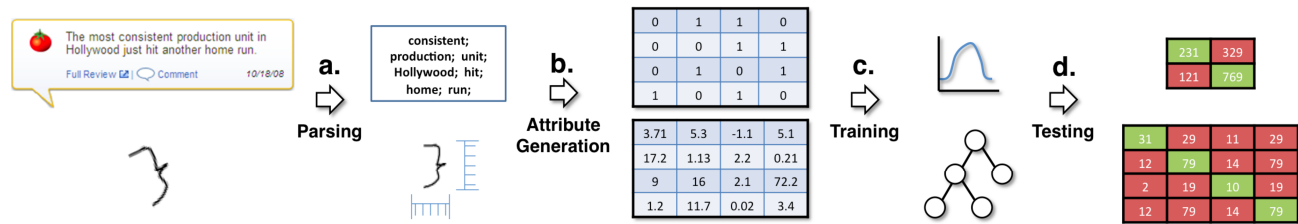


Figure 1: A wide variety of machine learning problems share a common classification pipeline. The pipeline describes how data is transformed into a model: (a) raw data is parsed, (b) attributes are computed from parsed data, (c) a model is trained, and (d) the model is tested. This figure above shows two problems, sentiment analysis and gesture recognition, that share the same pipeline.

THE MACHINE LEARNING PROCESS

Gestalt supports two high-level tasks in applying machine learning: *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 1 shows two example pipelines, in which: (a) data is transformed into discrete examples, (b) attributes[†] are computed over each example, (c) a learning algorithm is used to train a model, and (d) the accuracy of that model is evaluated. Not all pipelines are identical, but their structure is similar: a linear progression of computation transforms data into a model that can be experimentally evaluated.

Analysis allows developers 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, attributes, and model output [16]. In addition to final model output, this requires examination of intermediate data to ensure that each step in the pipeline behaves as expected. Developers examine whether data is correctly parsed and discretized, whether attributes 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 developer's next implementation action. Developers often revisit prior steps, such as collecting additional data, debugging implementation of attributes, brainstorming new attributes, or reconsidering their modeling 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.

PROVIDING GENERAL-PURPOSE SUPPORT

This section introduces two canonical machine learning problems: movie review *sentiment analysis* and pen-based *gesture recognition*. We discuss important differences between these problems, as these differences illustrate a range of support needed in a general-purpose tool. We then

discuss their similarity, as their common structure provides the basis for Gestalt's integrated support.

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.* [15]. Following Pang *et al.*'s process, a developer collects positive and negative movie reviews, formats reviews to plain text, and computes word-count attributes (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 [17]. A developer 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 learned 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 attributes (e.g., the length of the stroke, measures of angles in the stroke). Cross-validation experiments then evaluate the pipeline.

Problem Differences

Sentiment analysis is a *two-class* problem, but 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 developer 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 developer 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. They are still human verifiable, but require more attention than a gesture.

[†] We avoid the overloaded term *feature*, which could refer to either an *attribute* of data or a *capability* of Gestalt. Both alternatives are descriptive, though not as commonly used as the word *feature*.

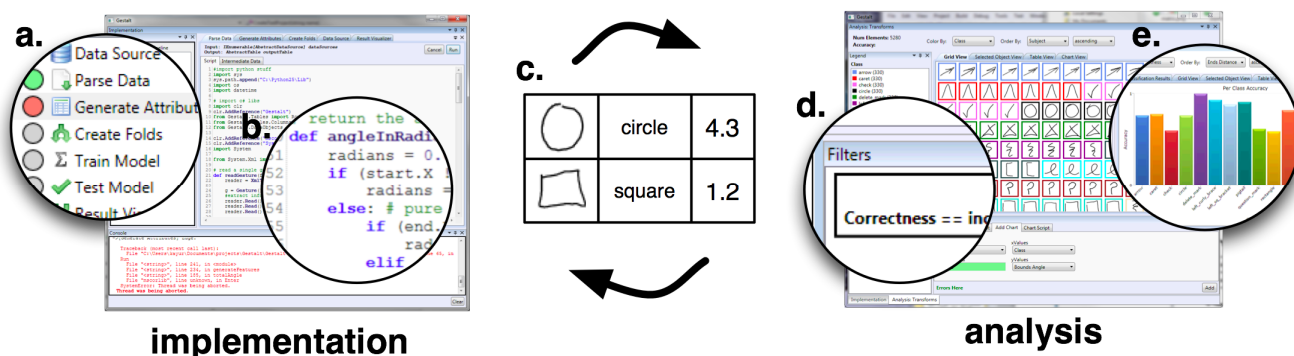


Figure 2: The implementation perspective provides developers with structure through its classification pipeline view (a) and flexibility by allowing them to write code to represent their specific problem (b). A common data structure (c), shared between analysis and implementation, allows developers to quickly switch between the two tasks. The analysis perspective allows developers to interact with the provided visualizations (e) by filtering, sorting, and coloring (d).

These problems also illustrate differing *interpretability* of their attributes, including *verifiability* and *sparseness*. Individual values of sentiment attributes are easier to *verify*. A developer can quickly check the value of a word-count attribute 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. On the other hand, sentiment attributes are *sparse*. Each review has a large number of attributes, most word-count values are zero, and only non-zero values have an effect on the final model. The gesture recognition problem is defined by a small set of dense attributes, where each attribute may have a distinct value and an effect on the final model.

A final difference we emphasize is how solutions 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 often be misleading. People may differ in how they draw a gesture, and random sampling ignores this lack of independence. 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 used. Models are trained with data from all but one person, then tested with data from that person.

Problem Similarity

Although these problems are different at nearly every step, Figure 1 shows there is a similar structure to their classification pipelines. Both separate data into discrete examples, compute attributes describing each example, and conduct experiments that identify sets of examples that are correctly or incorrectly classified by the pipeline. This common structure provides leverage for a general-purpose tool. In our development of Gestalt, we have examined how an integrated environment can provide necessary flexibility at every stage of a process while also leveraging this common structure to make developers more effective in their application of machine learning. The next section introduces Gestalt and its capabilities. In later discussion, we consider limitations of our current implementation and the general-purpose approach.

GESTALT

Developers interact with a classification pipeline in Gestalt through two high-level perspectives: an *implementation* perspective and an *analysis* perspective (Figure 2). This parallels the common distinction between coding and debug perspectives in modern development environments (e.g., Eclipse, Microsoft Visual Studio). The implementation perspective allows developers 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 developers as they implement a pipeline, analyze data, and transition between these perspectives.

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 the parts a developer needs to interact with to create a solution. For example, Crayons allows developers to input data and see the output of a model, but provides no control over the attributes or the learning algorithm [8]. Crayons achieves its ease of use by cloaking this complexity. However, it is thus 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 Gestalt is that *general support cannot be achieved 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 2a). Gestalt preserves *flexibility* by defining each step using IronPython scripts written in a built-in text editor (Figure 2b). This combination provides an explicit structure without constraining what a developer can do in that structure. Gestalt thus provides the same flexibility as general-purpose programming environments (e.g., Eclipse, MATLAB).

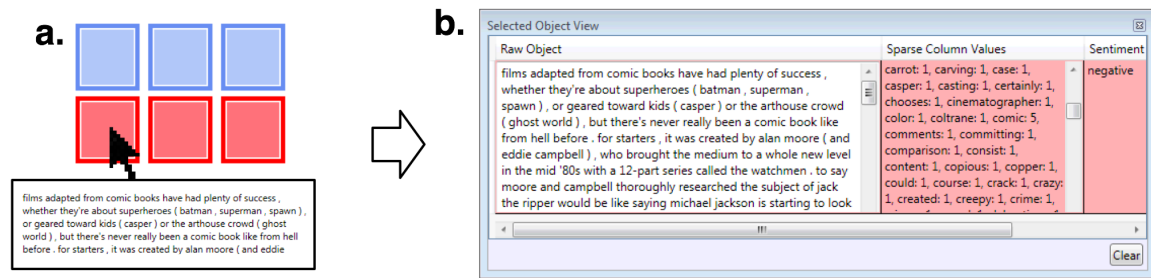


Figure 3: By looking at the raw data next to the attributes computed from that data, developers can gain a better understanding of system behavior. Here a developer is shown a thumbnail of movie review data (a). The developer clicks on the thumbnail to examine the raw data, attributes computed from it, and the fact that it is currently misclassified (b).

Gestalt's explicit structure provides a basis for its other functionality. For example, Figure 2a shows how Gestalt can help developers 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, 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. This section discusses how each step can be used as a launching point for analysis, helping developers better understand the behavior of their system through inspection of the input and output at each step.

Appropriate Data Structures

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 often hide seedy details of this portion of a machine learning system. But data comes in many forms and sizes, so *effective data management is a requirement for general 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 attributes. 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 modeling algorithms, Tableau focuses on powerful visualizations of tabular data). Despite their common tabular nature, data representations in such tools are not identical. Developers 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 attributes generally lack support for examining the original data used to compute those attributes (the raw data is typically not propagated forward by the attribute generation script).

Gestalt's use of a *single unified table* means *developers are freed from managing data conversion or moving data between tools*. This is critical to enabling fluid and easy movement between interpretation and analysis. Gestalt's

data representation implements several enhancements to a standard table. First, attribute columns are typed and tagged according to where they are used in the classification pipeline. All attributes can be used to summarize, visualize, and interact with data, but only some of those attributes can be used to build a model. Tagging of columns allows Gestalt to track which attributes should be used by a learning 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 attributes found in many problems (e.g., sentiment analysis).

Visualizing and Aggregating Examples:

How do I see my data?

Developers reason about system behavior by examining data and its relationship to attributes and classification. Domain-specific tools generally include a visual component that provides this feedback. This allows developers 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 learned 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 separating the logic needed to view *one example* from the logic to combine many single examples into an aggregate view. Developers can write code to visualize an example, and Gestalt then integrates that into aggregate visualizations throughout the pipeline. Two examples of aggregate visualizations are the grid view (Figure 3a, 4a, 4c) and the table view (Figure 3b).

We also 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 developers can use aggregate views to gain insight into their data. Gestalt's emphasis on code-based *flexibility* allows developers to adapt those visualizations to meet the needs of their particular data.

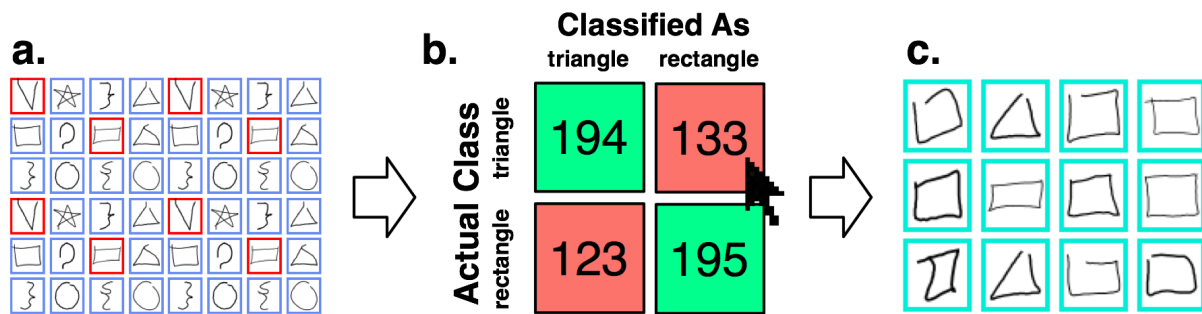


Figure 4: In Gestalt developers can use faceted browsing techniques to understand data. Here a developer tries to understand why triangles are confused with rectangles by filtering the full set of examples (a) through a click on a confusion matrix cell (b). The filtered examples (c) show that the confusion is due to mislabeled data.

Interactive, Connected Visualizations

How can I relate my data, attributes, and results?

Grouping and summarizing examples can help a developer understand a classification pipeline. Gestalt’s analysis emphasizes *interactive visualizations*, inspired by work in interactive visualization tools [4]. Support is provided for faceted browsing, filtering, sorting, and coloring examples. Grouping and summarization operations can be applied according to attribute values, according to columns added to examples by steps in the classification pipeline, and according to tags added to examples by a developer.

Gestalt’s support for machine learning goes beyond such prior general-purpose visualization tools by *connecting data generated across the entire classification pipeline*. In the case of domain-specific tools, consider that 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 developers understand relationships between data, attributes, and results.

Figure 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 developer 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 attributes, and its classification. Pulling this into a single view allows a developer 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 presents an example of a developer 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 developer might apply a filter to isolate examples that have a particular attribute value. Examining these might suggest a possible bug in the code computing that attribute. Connected visualizations allow developers to quickly assemble the information needed to examine such questions.

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’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: *developers can more quickly implement and analyze different potential versions of a machine learning system*. 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 developers to quickly analyze the important parts of their system.

CURRENT TOOL WORK

Several categories of tools can be used in machine learning applications and warrant discussion with regard to Gestalt.

Domain-Specific Tools

Domain-specific tools support both implementation and analysis, but do so at the expense of flexibility. For example, Crayons supports the learning of models that segment images [8]. A developer captures an image and colors regions that correspond to different segments. The system learns a model from these labeled pixels, and the developer analyzes the model’s performance by applying it to new images and overlaying the results on those images. The designer iterates by correcting model mistakes, thus providing new data for the classification pipeline. Crayons achieves this ease of use by limiting flexibility. Input is limited to providing more training examples, and analysis is limited to looking at classification results overlaid on images. Developers cannot access other information that might help them iterate (e.g., attribute values).

Domain-specific tools have been created for a variety of problems, including computer vision systems [14], simple sensor-based recognizers [11], and interactive concept learning in image search [9]. Because the number of

domains affected by machine learning is large and growing, designing domain-specific tools for each is untenable. Domain-specific tools often target non-programmers, who are unlikely to be able to make major changes to the inner workings of a system. Gestalt targets developers and can take a different approach. Gestalt focuses on providing the necessary development support to make implementation and analysis easier for a wide variety of domains. We are thus lowering the barrier to using machine learning, so that the large population of developers can join the ranks of expert researchers in their ability to apply machine learning.

Disconnected General-Purpose Tools

A variety of general-purpose tools support either implementation or analysis of machine learning systems. Weka is a well-known example, providing developers with a large library of machine learning algorithms [20]. Interactive visualization tools like Tableau can be applied to data exported from machine learning systems [19].

Tools that each support a portion of the machine learning pipeline create gaps that are a fundamental obstacle to effectively moving between implementation and analysis. Developers must explicitly choose to move from one tool to another, typically losing any established working context. It is entirely upon the developer to bridge the gaps between tools: writing custom scripts to convert between data formats exported by different tools, aggregating and visualizing raw data, storing and linking intermediate information computed throughout the pipeline. For example, a canonical pipeline for the sentiment analysis problem might use Python to process reviews and obtain word-count attributes, then Weka to train a model, then Tableau to analysis experimental results. Reproducing the interactions from Figure 3 and Figure 4 would require extensive developer effort. Gestalt connects steps, aggregates examples, and enables interactivity to allow developers to focus on the logic of their pipeline and analyses of how data is transformed in that pipeline.

Connected General-Purpose Tools

Connected general-purpose tools are capable of addressing the entire classification pipeline. These can be further decomposed into *dataflow* and *programming* environments.

Dataflow environments provide sets of discrete components that can be combined to implement desired behaviors [2]. Some dataflow tools even provide components targeting machine learning problems [10]. Dataflow tools generally focus on using pre-built components, so it is relatively difficult to create new components or modify the behavior of existing components. In contrast, machine learning problems vary in behavior. The structures of the sentiment analysis and gesture recognition problems are similar, but the behaviors of steps for data parsing and attribute generation are very different and unlikely to be provided as part of any standard set of prebuilt components. Gestalt’s focus on developer flexibility, critical to allowing rapid iteration on a pipeline, is more similar to the support provided by general programming environments.

Modern general programming environments work well for writing code that describes the behavior of a program, but are not designed for writing code that *learns* from data. Many people experienced in the application of machine learning report a preference for MATLAB, because it provides better support 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 provide sufficient functionality to significantly 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 developers face when using machine learning. Developers 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 has not been designed for machine learning, and all elements in a matrix are of a single datatype. Developers therefore must maintain multiple parallel matrices to store raw data, numerical attributes, string attributes, and attribute 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, developers would need to rewrite most of the functionality provided by Gestalt within MATLAB.

EVALUATING BUG FINDING IN GESTALT

Our study compared bug-finding performance for participants using Gestalt with a baseline condition similar to MATLAB. Prior research shows the developers consider connected environments, like MATLAB, to provide the best support for the machine learning process [16]. This section describes our baseline system, the tasks in our study design, and the major results of our experiment.

Participants

We recruited 8 participants (2 female) for our 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 supervised machine learning. This population is consistent with the target audience of Gestalt: software developers who know how to apply machine learning.

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. We provided an API with which could be used to reproduce all of Gestalt’s visualizations.

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. Participants had to write code to connect raw data, attribute values, and classification results or to create side-by-side visualizations.

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. We chose this study design, instead of a design that compared Gestalt directly to MATLAB, because we wanted to increase our confidence that any differences we observed were due to the capabilities we had taken away (and not other differences in the tools, such as the syntax of the programming language).

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 (e.g., ordering or pairing of interface and task), we counterbalanced the task with condition (Gestalt and baseline) and order (first and second).

Our 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.

We 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, we focus on such measurements as the time spent in various visualizations over the entire study.

Sentiment Analysis and Gesture Recognition Tasks

Participants built solutions for the two problems discussed earlier: sentiment analysis and gesture recognition. Each contained data and five scripts: `parsing`, `attributes`, `splitting`, `training`, and `testing`. We created working solutions for both Gestalt and the baseline, then injected five bugs into each solution. The machine learning code for the baseline and Gestalt differed only in how scripts were called and how data was maintained between steps. These factors were intrinsic to the differences measured in our results. Although we have described the two problems previously, we provide additional details about their implementation.

The sentiment analysis task classified movie reviews as positive or negative. We used 1,000 negative and 1,000 positive reviews from a standard sentiment analysis dataset

[15]. We computed word-count attributes, built a Naïve Bayes model, and evaluated using three-fold cross validation. After building a working system, we introduced the following bugs into the sentiment analysis problems:

- S1:** mislabeled 300 positive and 300 negative examples [`data`]
- S2:** positive examples are read in twice [`parsing`]
- S3:** instead of removing stop words, the code removes everything except for stop words [`attributes`]
- S4:** only updates the count for one attribute [`attributes`]
- S5:** each fold tests on the training set [`splitting`]

The *gesture recognition* task involved building a model that classifies a pen-stroke as one of 16 different gestures. We used a standard dataset of 5280 gestures collected from 11 different people [21]. We normalized strokes, computed attributes, built a Rubine model, and evaluated using per-person cross-validation. We introduced the following bugs:

- G1:** mislabeled gestures (30 triangles swapped with rectangles, 30 circles with stars, and 30 carets with checks) [`data`]
- G2:** (x, y, t) points are loaded as (t, x, y) [`parsing`]
- G3:** does not load all of the examples [`parsing`]
- G4:** sine and cosine values are the same for one of the attributes [`attributes`]
- G5:** tests on the same person in each fold [`splitting`]

We 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 [15]. The cross-validation bug in our gesture recognition task is the same one reported by Hodges and Pollack in their work [12]. Other bugs were based on common mistakes, such copy-paste errors [13].

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 attribute 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 our task, repairing 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). We then pointed them to a directory containing correctly labeled data.

Finally, because the inserted bugs interacted with each other, the accuracy of the classifier could increase or decrease erratically (even going above the target accuracy).

This was a deliberate choice; erroneous 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.

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 developer. The document detailed the steps taken by the previous developer, and participants were informed the developer 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 we were studying the effect of Gestalt’s novel capabilities and not the usability or learnability of the system, participants were told they could use the experimenter as an intelligent help system during the task. This included asking questions about APIs, visualizations, the machine learning problems, and error messages.

Participants were asked to talk aloud, describing their progress in the bug finding process. Participants were told the experimenter might ask questions about their state or current action. We asked participants to think aloud about the 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.

RESULTS

Participants unanimously preferred Gestalt and were able to find and fix more bugs using Gestalt than using the baseline. Figure 5 shows an overview of bugs per condition. To examine our *found* and *fixed* measures, we conducted a mixed-model analysis of variance. We modeled *participant*

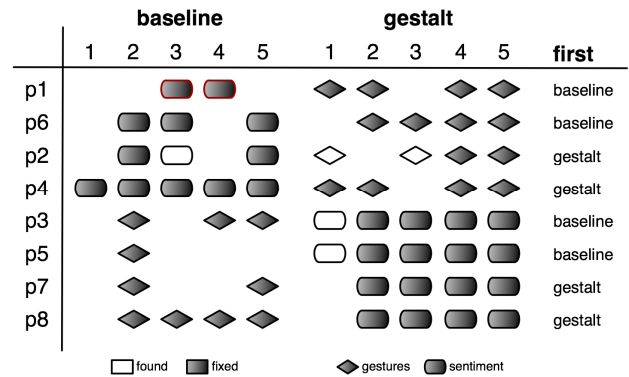


Figure 5: Developers *found* and *fixed* significantly more bugs in the Gestalt condition.

as a random effect and modeled *condition* (Gestalt vs. baseline), *task* (sentiment analysis vs. gesture recognition), and *trial* (first vs. second) as fixed effects. We also modeled the interactions *condition*×*trial* and *condition*×*task*. We used these same independent variables in all of the analyses we report in this section.

We 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. We verified the interaction *condition*×*trial* was not significant ($p > .42$), confirming the effectiveness of our counterbalanced design. Participants in the Gestalt condition *found* significantly more bugs (4.25 vs. 2.88 bugs, $F_{1,5}=11.42$, $p \approx .019$).

We 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 our counterbalance effectiveness by verifying the lack of significant interaction *condition*×*trial* ($p > .72$). Participants in the Gestalt condition *fixed* significantly more bugs (3.75 vs. 2.75 bugs, $F_{1,5}=7.27$, $p \approx .042$).

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. We ground our observations in free response questions from our questionnaire and secondary measures of performance. We also discuss limitations of our study, Gestalt’s implementation, and general-purpose tools.

The Importance of Structure

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

However, we 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.*”

Creating Individual Example Visualizations

Even though we provided standard visualizations of the individual examples in both conditions, some participants created 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) used this to plot a gesture’s stroke. This confirms developers can and will develop quick, simple visualizations of raw data when given proper tools. This is promising evidence for Gestalt’s approach of using developer-created visualizations of individual examples in aggregate visualizations to help developers understand data, attributes, and results.

The Need for Connectivity

Participants in both conditions actively tried to relate attribute values and results to their raw data. Gestalt’s connected visualizations make it easy to compare their data, attributes, and classification results. When taken away in the baseline, participants expressed frustration. One participant, who worked in Gestalt first, explicitly described that he wanted to see the raw data next to the attributes 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 combined table view; two did this before having used Gestalt. In all three cases, they opened two separate table views, one after parsing and one after attribute computation. They then resized these tables and placed them side-by-side so they could visually compare attributes with data.

Interactivity

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

Participants also used more *kinds* of views. In our post-study questionnaire, we asked participants to tell us which faceted search capabilities (e.g., filtering) and views they used (e.g., grid view). We 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 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 developers 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. He clicked in a confusion matrix to see examples of triangles classified as rectangles, then found the mislabeled examples.

Study Limitations

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

Our study was also limited to finding bugs in unfamiliar code. The 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. Participants found value in Gestalt’s pipeline structure. Their comments in the open-ended questionnaire lead us to believe Gestalt’s structure will also assist developers solving machine learning problems from scratch.

Our study focused on two problems for which developers had some intuition about the data. 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 attributes. Developers 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 developers to create individual visualizations embodying information to best help them to understand their data.

Limitations of Gestalt

Our 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. He then manually computed attributes (using pen and paper) and compared them to the values computed during attribute

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.

While Gestalt can be used to build machine learning systems for many domains, there are some problems 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 impacts the types of learning algorithms Gestalt supports, but also some of Gestalt's core capabilities. For example, our current grid and table aggregate visualizations cannot properly visualize relationships inherent to sequential data (e.g., time-series). It is also non-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. We 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.

CONCLUSION

Gestalt supports the entire *process* of applying machine learning: *implementing* a classification pipeline, *analyzing* data as it moves through that pipeline, and easily transitioning between these perspectives. We have discussed how Gestalt's capabilities generalize advances from prior domain-specific tools to provide general-purpose support. A comparison of participants using Gestalt with a baseline condition similar to MATLAB showed participants *find* and *fix* more bugs with Gestalt and that Gestalt's flexibility and visualizations were primary contributors to their success. These results show that helping developers understand relationships between the various steps in a classification pipeline is important to their success.

ACKNOWLEDGEMENTS

We thank Morgan Dixon, Joe Devietti, and Scott Saponas for their invaluable feedback. This work has been supported in part by MSR and NDSEG fellowships for Kayur Patel and by the National Science Foundation under grants IIS-0812590 and CCF-0952733.

REFERENCES

1. Ball, N.M. and Brunner, R.J. *Data Mining and Machine Learning in Astronomy*. 2009.
2. Blume, P.A. *The LabVIEW Style Book*. Prentice Hall 2007.
3. Candea, G. and Fox, A. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. *HotOS 2001*.
4. Card, S.K., Mackinlay, J.D., and Shneiderman, B. *Readings in Information Visualization: Using Vision to Think*. 1999.
5. Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. Path-based Failure and Evolution Management. *NSDI 2004*.
6. Consolvo, S., McDonald, D.W., Toscos, T., Chen, M.Y., Froehlich, J., Harrison, B., Klasnja, P., LaMarca, A., LeGrand, L., Libby, R., Smith, I., and Landay, J.A. Activity Sensing in the Wild: A Field Trial of UbiFit Garden. *CHI 2008*.
7. Cruz, J.A. and Wishart, D.S. Applications of Machine Learning in Cancer Prediction and Prognosis. *Cancer Informatics 2*, (2007).
8. Fails, J.A. and Olsen, D.R. Interactive Machine Learning. *CHI 2003*.
9. Fogarty, J., Tan, D., Kapoor, A., and Winder, S. CueFlick: Interactive Concept Learning in Image Search. *CHI 2008*.
10. Gil, Y., Ratnakar, V., Kim, J., Gonzalez-Calero, P.A., Groth, P., Moody, J., and Deelman, E. Wings: Intelligent Workflow-Based Design of Computational Experiments. *IEEE Intelligent Systems*, To Appear.
11. Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S.R. Authoring Sensor-based Interactions by Demonstration with Direct Manipulation and Pattern Recognition. *CHI 2007*.
12. Hodges, M.R. and Pollack, M.E. An 'Object-Use Fingerprint': The Use of Electronic Sensors for Human Identification. *UbiComp 2007*.
13. Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do Code Clones Matter? *ICSE 2009*.
14. Maynes-Aminzade, D., Winograd, T., and Igarashi, T. Eyepatch: Prototyping Camera-based Interaction Through Examples. *UIST 2007*.
15. Pang, B., Lee, L., and Vaithyanathan, S. Thumbs Up?: Sentiment Classification Using Machine Learning Techniques. *EMNLP 2002*.
16. Patel, K., Fogarty, J., Landay, J.A., and Harrison, B. Investigating Statistical Machine Learning as a Tool for Software Development. *CHI 2008*.
17. Rubine, D. Specifying Gestures by Example. *SIGGRAPH 1991*.
18. Simon, I., Morris, D., and Basu, S. MySong: Automatic Accompaniment Generation for Vocal Melodies. *CHI 2008*.
19. Stolte, C. Visual Interfaces to Data. *SIGMOD 2010*.
20. Witten, I.H. and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques*. 2005.
21. Wobbrock, J.O., Wilson, A.D., and Li, Y. Gestures Without Libraries, Toolkits or Training: A \$1 Recognizer for User Interface Prototypes. *UIST 2007*.