Mallard: Turn the Web into a Contextualized Prototyping Environment for Machine Learning

Xiong Zhang
University of Rochester
Rochester, NY, USA
xzhang92@cs.rochester.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

Machine learning (ML) can be hard to master, but what first trips up novices is something much more mundane: the incidental complexities of installing and configuring software development environments. Everyone has a web browser, so can we let people experiment with ML within the context of any webpage they visit? This paper's contribution is the idea that the web can serve as a contextualized prototyping environment for ML by enabling analyses to occur within the context of data on actual webpages rather than in isolated silos. We realized this idea by building Mallard, a browser extension that scaffolds acquiring and parsing web data, prototyping with pretrained ML models, and augmenting webpages with ML-driven results and interactions. To demonstrate the versatility of Mallard, we performed a case study where we used it to prototype nine ML-based browser apps, including augmenting Amazon and Twitter websites with sentiment analysis, augmenting restaurant menu websites with OCR-based search, using real-time face tracking to control a Pac-Man game, and style transfer on Google image search results. These case studies show that Mallard is capable of supporting a diverse range of hobbyist-level ML prototyping projects.

CCS Concepts

•Human-centered computing → Human computer interaction (HCI);

Author Keywords

contextualized machine learning, ML prototyping

INTRODUCTION

Machine learning (ML) is now a highly sought-after skill in areas such as technology, scientific research, healthcare, and public policy [40, 53, 65]. There is widespread demand for people who can acquire data in the wild, mold it to their needs, and use it to make predictions about the world. Over the past decade, software frameworks (e.g., TensorFlow [21], Keras [6]), cloud computing services (e.g., AWS, Azure, Google Cloud), and libraries of pretrained models (e.g., MobileNet [46], WaveNet [66], YOLO [60]) have given more people access to ML. However, novices still face significant barriers getting started with creating ML-based applications.

The math that underlies ML can be challenging to master [33], but what first trips up novices is something much more mundane: the incidental complexities of installing and configuring their software development environment with the requisite tools. Before even getting to write a single line of code, they must wrestle with software package managers, library dependencies, OS version incompatibilities, command-line environments, idiosyncratic data formats, or cloud computing dashboards. Novices often report such software setup struggles as barriers to getting started with ML [63, 64, 69].

Web-based interactive tutorials can lower such barriers by allowing users to write code within their browser without installing dependencies. While convenient, their main limitation is that they restrict users to whatever data and ML models the creator of each tutorial provides. For instance, a tutorial may embed a Python code editor connected to a server that hosts data sets and pretrained models that its creator selected. This is a good start, but what if learners could freely tinker with ML on authentic data they are personally motivated by rather than only following pre-made tutorials? Since the web is a rich source of motivating data, *what if people could experiment with ML within the context of any webpage they visit?*

To address this question, we developed Mallard, a browser extension that turns the web into an ML prototyping environment. Mallard is inspired by physical prototyping environments such as *breadboards* [39, 68] for electronic circuits[1]: Breadboards allow novices to quickly wire up, test, and play with electronic components without the overhead of soldering or fabrication. Likewise Mallard allows users to quickly acquire data from webpages they visit and connect ML components to them without the overhead of software infrastructure setup. Just like how one would not expect to ship production-grade electronics on a breadboard, we also do not expect Mallard to be used for production-scale deployment of ML apps.

Figure 1 shows how Mallard enables users to prototype ML-based applications entirely inside of the Chrome web browser so that analyses occur directly within the context of webpages that they are visiting. We call this a *contextualized* machine learning workflow. Users typically follow a four-step process:

[1] Mallard = **Ma**chine **l**earning **l**ightweight breadbo**ard**

UIST '19, October 20-23, 2019, New Orleans, LA, USA.

ACM ISBN 978-1-4503-6816-2/19/10 ...$15.00.
https://doi.org/10.1145/3332165.3347936

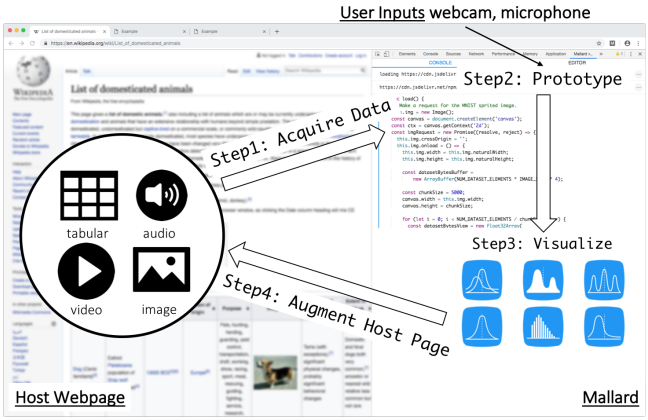


Figure 1: Mallard enables hobbyists to prototype machine learning applications directly within the context of existing webpages they visit.

1. **Acquire** data from existing webpages
2. **Prototype** in an interactive coding environment
3. **Visualize** debugging outputs in the console
4. **Augment** the host webpage with analysis outputs

Here is an example usage scenario: Alice is a teacher in a classroom where students get low-cost Chromebook laptops. Due to resource limitations, they cannot install ML tools on the laptops and cannot afford access to cloud computing resources. To teach her class about algorithmic bias [25], Alice installs the Mallard browser extension and starts prototyping:

1. She visits her school's private webpage, which contains all students' photos and profiles in an HTML table.
2. Mallard detects tabular data on that webpage and parses it into a JavaScript table object containing text and images.
3. In the Mallard console, she imports the TensorFlow.js JavaScript ML framework [64] and the face-api.js pretrained face detection model [54] by providing their URLs.
4. She writes JavaScript code in the console to run the face detection model on the imported student photos. After tuning the model, she tells Mallard to augment the webpage to display the model's predictions beneath everyone's photo.
5. When she demos this to her class, her students notice that many of their faces were not properly detected. Alice leads a discussion on algorithmic bias due to the face-api model's training data not matching her students' demographics.
6. Alice writes more JavaScript in the console to access her laptop's webcam and run the face detector on the captured live video feed. She saves her exploratory code and distributes it to all of her students' laptops.
7. Now every student can use their own webcams to play with the face detector and see who is getting properly detected.
8. Her class decides to retrain this model using Alice's webcam to take photos of their faces to use as new training data. As they do so, they notice more photos on the school webpage now being properly detected (even for students in other classes). Along the way, Mallard shows ML diagnostic visualizations to help them debug their model.

This scenario shows how Mallard serves as a browser-based breadboard for connecting ML frameworks and pretrained models to data on live webpages. If Alice did not have this system, she would need to set up a full development environment on her machine or in the cloud, then help all of her students do so. She would also need to figure out how to scrape private website data using her school's custom login credentials and parse that data into a programmable form. Then she must create a separate GUI to display the face detection algorithm's outputs alongside student profile photos. With Mallard, everything happens within the original school webpage.

This paper's main contribution is the idea that ***the web can serve as a contextualized prototyping environment for machine learning*** by enabling analyses to occur directly within the context of data on actual webpages rather than in isolated standalone environments. We realized this idea by building Mallard, a browser extension that scaffolds acquiring and parsing web data, importing and writing code that operates across multiple pages, and debugging using visualizations.

The web is a compelling prototyping environment due to its ecosystem of millions of instantly-importable JavaScript packages on npm, GitHub, and other websites [17]. In addition, GPU-accelerated JavaScript engines make it possible to efficiently run ML models in the browser with frameworks such as TensorFlow.js [64] and ml5.js [4]. The web also contains huge amounts of motivating data [35, 72] ranging from Wikipedia tables to government CSV files, along with billions of more images and videos to power ML apps. Just like how breadboards make it easy for novices to tinker with connecting electronics components together to build prototype circuits, Mallard aims toward a future where hobbyists, students, and educators can tinker with ML models without getting bogged down by software infrastructure complexities.

To demonstrate the versatility of Mallard, we performed an informal case study where we used it to prototype nine ML-based applications spanning a variety of domains and interaction types: 1) augmenting Amazon and Twitter websites with text sentiment analysis, 2) augmenting restaurant menu websites with OCR-based search, 3) classifying birds on stock photo websites, 4) augmenting Wikipedia tables with animal species labels, 5) retraining a neural network for binary classification, 6) training a face recognizer from web images, 7) live coding on an ML tutorial webpage, 8) using real-time face tracking to control a Pac-Man game, 9) style transfer on Google image search results. Most took a few dozen lines of JavaScript code to create custom UIs and to hook into existing ML frameworks and models. These case studies show that Mallard is capable of supporting a range of breadboard-style ML projects that might appeal to educators or hobbyists.

The contributions of this paper are:

- The idea that the web can serve as a contextualized prototyping environment for machine learning by unifying code and data within a domain-specific browser developer tool.
- Mallard, a system that instantiates this idea with scaffolding to help users acquire data, work with pretrained ML models, and augment host webpages with ML outputs.

RELATED WORK

Mallard continues the long lineage of work on lowering barriers to machine learning (ML) but is unique in its use of the web as a contextualized prototyping environment for ML.

Domain-specific ML tools such as Crayons [41] and Eyepatch [51] provide direct manipulation interfaces to help users prototype specific types of ML applications: image classifiers and computer vision from video streams, respectively. More general-purpose ML tools such as Weka [44] enable users to import tabular data and train a wide range of models using a GUI. These systems strive to lower barriers for novices by not requiring them to write any code to train and test basic models. In contrast, Mallard is a full JavaScript-based coding environment, which has a higher barrier to entry but enables more sophisticated and custom ML apps to be built. Mallard also uses webpages directly as both the input data and an output canvas, whereas these tools require users to acquire, clean, and import data into a standalone environment and then manage outputs separately. Finally, Mallard is mainly for prototyping with pretrained ML models along with light retraining on new data, so it does not have scaffolding for advanced label and data management to train new models from scratch.

ML-focused programming environments often provide scaffolding such as interactive visualizations to help users debug their pipelines. For instance, Gestalt [57] shows visualizations of data as it moves through the ML pipeline; users write custom code to visualize individual data points, and Gestalt uses that code to create aggregate visualizations. Computational notebooks such as Jupyter [10], Colaboratory [5], Observable [9], and Iodide [37] allow programmers to interleave code with debugging visualizations as they are developing ML apps. Visual analytics tools such as TensorBoard [13], What-If Tool [20], Manifold [71], Prospector [49], and ModelTracker [24] can augment programming environments with ML-specific visualizations.

Mallard similarly allows users to interleave visualizations with code and comes with a set of simple ones for profiling and debugging ML model performance. However, Mallard differs in a fundamental way from all of these existing systems since it uses webpages directly as data substrates, which removes the friction of needing to import data into a standalone coding environment. Note that even web-based computational notebooks (e.g., Jupyter) and web IDEs (e.g., repl.it [12], Cloud9 [3]) are still standalone coding environments; programmers must import data into them either via cloud APIs or by downloading and managing data files.

ML libraries and frameworks can lower development barriers by encapsulating common code for wrangling input data and training and testing ML models. General-purpose libraries such as scikit-learn [58] for Python and MLlib [52] for the Apache Spark ecosystem contain a wide array of ML-related algorithms. More specialized frameworks such as TensorFlow [21], Caffe [48], Theano [23], PyTorch [56], and Keras [6] enable users to construct neural nets for deep learning applications. In recent years, JavaScript-based ML frameworks such as TensorFlow.js [64], Keras.js [36], and ml5.js [4] have grown more popular as JavaScript engines in web browsers have become faster and taken better advantage of increasingly powerful GPUs. Potential benefits of running ML in the browser include easier distribution to end users, lower latency for interactive ML apps, and better privacy since personal data does not need to be sent off the user's device [38]. Mallard complements this growing trend of JavaScript-based ML frameworks by augmenting them with a browser-native programming environment.

Using webpages as data substrates is appealing due to the large amounts of structured and semi-structured data on the web. For instance, a web crawl that Google made in 2008 found over 154 million HTML tables containing relational data [32], an amount which has likely grown exponentially over the past decade. Browser-based tools can thus scrape, reformat, and visualize data directly on existing webpages. For instance, Sifter [47] augments structured data on webpages with filtering and sorting controls. Mashup tools such as Marmite [70] and Vegemite [50] enable users to extract tabular web data into a spreadsheet to be combined with other data sources. Vispedia [34] lets users augment tables, lists, and infoboxes on Wikipedia pages with custom data visualizations. These systems do not require users to write code, which makes them easier to use but limits their scope. Mallard continues this tradition by exposing a full JavaScript-based programming environment with affordances for prototyping ML.

Mallard is also inspired by end-user programming tools for the browser such as Chickenfoot [27] and Greasemonkey [59] that allow users to alter behaviors of opened webpages by writing JavaScript code in a browser sidebar editor. While in theory these could be used for prototyping ML applications, they lack important ML-centric scaffolding for workflow operations such as acquiring data, visualizing debugging outputs, and augmenting host webpages.

The closest related system to Mallard is our own DS.js [72], which parses tabular data on existing webpages and allows users to write JavaScript data science code to operate on that structured data using a specialized API. Mallard was heavily inspired by the high-level goal of DS.js to turn webpages into programmable substrates. However, DS.js is limited to only parsing tabular data on a single webpage and having users manipulate and visualize largely-numerical data within those static tables. Mallard extends the contextualized web prototyping ideas of DS.js by: 1) providing hooks to augment webpages with both enhanced styling and interactivity, which is important for ML demo apps, whereas DS.js just used webpages as static tabular data sources, 2) adding scaffolding for multimedia data (e.g., images, videos), which also leads to richer ML demos, 3) supporting mashups of data from multiple webpages, whereas DS.js was a single-page tool. We view Mallard as a generalization of DS.js's nascent ideas to open up lower-barrier prototyping possibilities for ML.

MALLARD SYSTEM DESIGN AND IMPLEMENTATION

Mallard is a developer tools extension for the Chrome web browser. The user activates it by opening the developer tools panel. We recommend positioning this panel as a sidebar with the currently-opened webpage (called the "host webpage")

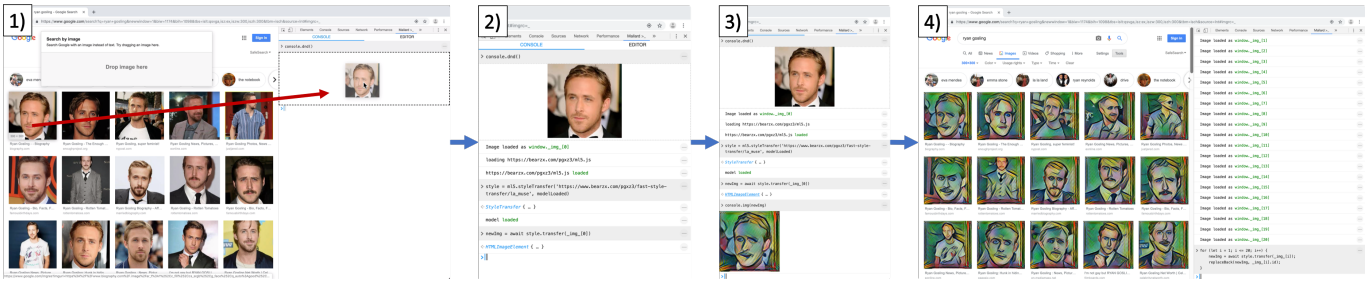


Figure 2: Example Mallard workflow from Case Study 9 of our evaluation: 1) acquiring image data via drag-and-drop, 2) writing code to apply a style transfer [42] ML model to that image, 3) seeing visual previews in the console, 4) applying that model to modify all images on the host webpage.

on the left and Mallard on the right (Figure 1). The Mallard panel contains an enhanced JavaScript console. However, unlike the built-in console, which runs code within the scope of the current webpage, code that the user writes in Mallard runs in the Chrome extension's sandboxed environment, which is separated from any individual webpage. This is critical both for providing isolation and for enabling Mallard to access data across multiple webpages so that users can create multi-page interactive apps (see Case Studies for examples). Mallard contains 5184 lines of JavaScript code built upon the React framework [11] and the jsconsole [62] project.

Here is how Mallard supports users in prototyping ML apps using the four-step workflow introduced in Figure 1. We will use Figure 2 as a running example to illustrate each step.

Step 1: Acquire Data from Existing Webpages

Mallard provides interactive scaffolding to let users acquire data from webpages they visit so that they can prototype ML explorations within the context of data that is personally meaningful to them. This scaffolding eliminates the burdens of writing data parsing code, managing data files separately on disk, and wrestling with login/authentication APIs (important for accessing private data such as student profiles within a school webpage). In short, if the user can reach a webpage, they can start prototyping ML within it.

When the user sees data of interest on a webpage, they can either right click on it and select "Import Data" or drag-and-drop it into the Mallard console. Mallard parses that data and exposes its resulting object as a JavaScript variable. Figure 3 shows how it recognizes common data types used in ML:

- **Multimedia data such as images, videos, and audio clips** contained within the pervasive HTML `<img>`, `<video>` and `<audio>` tags. Multimedia data is useful for a wide variety of ML apps such as image classification, face recognition, and speech recognition. ML frameworks usually have their own API calls that convert standard image, video, and audio DOM elements into framework-specific objects. For instance, TensorFlow.js [64] has a `fromPixels()` method that converts an image into a 2D tensor that can be fed into neural networks. Mallard can directly connect to those APIs with one line of code, but for frameworks without APIs to parse multimedia DOM elements, the user can write a custom handler to convert them into framework-specific objects. In Step 1 of Figure 2, we drag a celebrity photo from Google Image Search into the Mallard console, which turns it into an `<img>` object that can be further converted to a TensorFlow object.

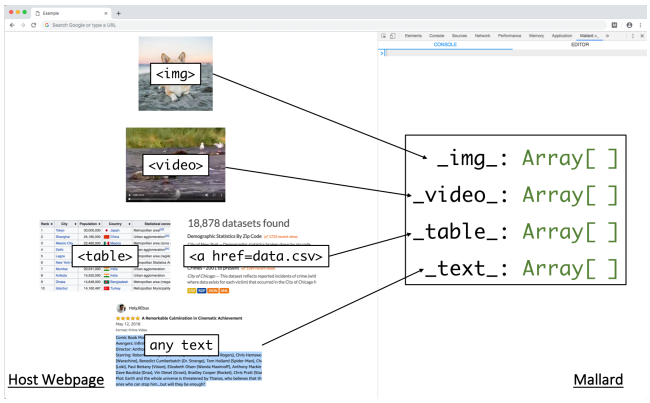


Figure 3: Mallard allows users to drag-and-drop common types of data found on webpages into an interactive console in the browser developer tools pane. It automatically parses that data into JavaScript objects so that users can write code to process them.

- **Structured tabular data** such as HTML tables and .csv/.tsv data files that appear as links on the current page. Mallard parses each into a JavaScript data frame object that resembles a relational database table. If the user clicks on an HTML table cell, Mallard detects the nearest-enclosing `<table>` tag; this heuristic was inspired by DS.js [72] and works well for flat tables that contain relational-style data, which is common on the web [32]. However, our parser does not support nested tables. The HTML parser strips off all styling information from text and uses the appropriate multimedia parser (see above) to turn `<img>`, `<video>` and `<audio>` tags into JavaScript objects.

- **Unstructured plain text**, which the user can highlight and import into the console as a standard JavaScript string.

- **JavaScript objects** within the current webpage. In some cases, webpages contain hidden state stored inside of JavaScript objects. The user can run a special `:acquire ${variable name}` command in the Mallard console to import that variable's current value. This step is necessary since Chrome extensions run in a separate environment from the host webpage's JavaScript.

For more complex parsing, the user can write custom JavaScript code to walk the host webpage's DOM and extract the desired elements. (One could also imagine pairing Mallard with PBD-based web scraping tools [35].)

After Mallard acquires a piece of data using one of the methods described above, it prints a message to the console to indicate the variable name that it automatically assigned to the resulting JavaScript object. Each data type appears in its own array, as shown in Figure 3. For instance, imported images get assigned variable names of `_img_[0]`, `_img_[1]`, `_img_[2]`, etc. The user can now write JavaScript code in the Mallard console or code editor to operate on these objects.

Finally, Mallard exposes the browser's built-in webcam and microphone inputs to the user, which are useful for prototyping video- and audio-based ML apps, respectively. Running the special `console.webcam()` and `console.mic()` scaffolding functions will pop up inline video/audio recording controls in the console without the user needing to write boilerplate setup code.

Step 2: Prototype in an Interactive Coding Environment

Mallard turns the browser into a lightweight IDE by providing a JavaScript code editor and enhanced console capable of displaying arbitrary HTML outputs (Figure 4). It unifies both the user's code and data in one place (the browser extension), which eliminates the need to install and configure a separate programming environment. A typical workflow involves prototyping in the console and copying selected snippets into the code editor for posterity. Mallard's console natively supports multi-line code inputs with syntax highlighting, which makes it easier to prototype entire functions at once. For example, in Step 2 of Figure 2, we wrote JavaScript in the console to import the ml5.js framework, use its API to load an image style transfer [42] model, and apply it to the imported image.

Loading external modules: To give users access to the entire ecosystem of JavaScript libraries available on npm, the canonical package management system currently with over 836,000 libraries [17], we implemented a `:load` console command. By calling it with the name of an npm module, Mallard will load and import that module from npm using the unpkg [18] service. Also, calling `:load` with a JavaScript file's URL from anywhere on the web will load its code directly. Popular JavaScript libraries for data cleaning, machine learning, and data visualization can be imported this way.

Saving and restoring state: The user can save the current state of their session to a JSON file, which can then be loaded by other users who have the Mallard Chrome extension installed. This saved state includes the URLs of all currently-opened pages and the contents of the Mallard code editor and console history associated with each URL. (Note that due to how Chrome developer tools works, there is a separate instance of the Mallard code editor and console for each page.) Now when another user loads this JSON state file, it will direct their browser to open all the stored URLs and load the saved code in the Mallard editor and console history for each URL. If a page requires the user to log in, they can enter that information directly in their browser. This way, one user can prototype an ML application that accesses personal data (e.g., within one's Instagram or Twitter account) and other users can run it on their *own* private data in their own browsers.

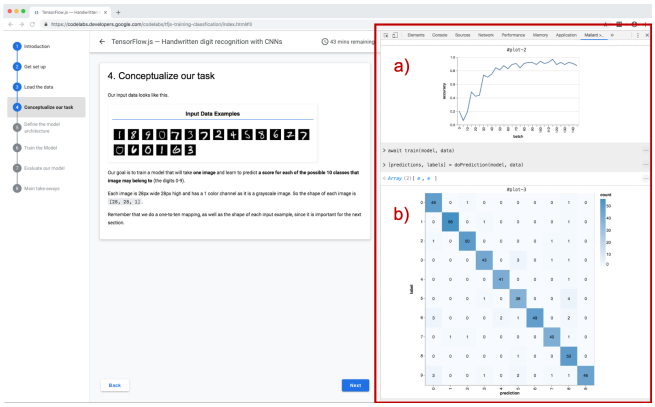


Figure 4: As the user follows a tutorial webpage for TensorFlow.js, a JavaScript ML framework [15], they can play with its sample code within the Mallard console (red box) and see ML debugging visualizations such as a) training accuracy curves and b) confusion matrix.

By default Mallard saves only the user's code and not the live in-memory state of JavaScript objects. However, there may be times when a user wants to save a computed object for others to access. This may occur when they have trained an ML model on their own personal data on a given website and want to share only that model with others but not to expose their raw personal data. To do so, they can run a special `:tojson ${variable name}` command on the Mallard console to serialize that model's object into JSON to save as part of their session's state. Many data structures within ML frameworks have JSON serialization methods, and users can write custom methods for those that do not.

Step 3: Visualize Debugging Outputs in Console

The Mallard console can display arbitrary HTML as the result of running code from either the console or editor. This capability is useful for showing multimedia outputs or data visualizations during the process of interactively debugging ML pipelines. For example, in Step 3 of Figure 2, we can see a live preview of the image in the console after it has been processed by the style transfer ML model.

When the user runs a snippet of code whose return value is a multimedia type that Mallard recognizes, it will render inline in the console. If the user wants a more persistent visual output, they can run an `:output` command to create an HTML output block in the console and repeatedly write into it. This is useful for creating a data animation that incrementally updates as an algorithm gets refined, such as seeing a linear regression line gradually get closer to fitting the training data.

ML-specific debugging visualizations: Although users are free to create arbitrary data visualizations using JavaScript libraries such as D3 [29] and Vega [19], Mallard comes with a set of visualizations commonly used in ML prototyping:

- **Histogram of training set classes** shows the distribution of training data seen so far, divided by class (e.g., cats,

dogs, and bears for an animal classifier). This lets users see which classes they need to collect more samples for.

- **Barplots of predicted classes** along with their prediction confidence levels, for multiclass classification (e.g., this test image is 85% likely to be a cat, 10% dog, 5% bear).
- **Confusion matrix** shows a 2D matrix of classes with the number of data points that were predicted in each class, along with its ground truth actual class (e.g., how many cats were mispredicted as dogs).
- **ROC curves** (Receiver Operating Characteristic) show true positive vs. false positive rates for various threshold value settings in binary classification tasks [45].
- **Accuracy curves and loss curves** plot model performance versus number of iterations, often used when training models such as neural networks. The shapes of these curves can indicate possible underfitting or overfitting [31].
- **Image and video overlays** enable custom renderings of bounding boxes and labels atop images and videos, used to show outputs of image/video recognition models. Figure 5 shows a screenshot from one of our case studies where we ran OCR to recognize text in restaurant menu images. We used the overlay scaffolding to render orange highlights when the user searches for text that appears in the images.

These visualizations can be accessed via function calls. Figure 4 shows ML debugging visualizations from one of our case studies: as the user follows a tutorial to train a JavaScript neural network for handwriting recognition [15], Mallard is open on the right. As they tune the model, they can see the accuracy curve and confusion matrix in Mallard's console.

To implement each visualization, we wrote around a dozen lines of code to hook it up to the ML frameworks used in our case studies (see Table 1 for details). This is necessary because each framework exposes its model training and inference data structures using its own APIs. Supporting other frameworks would likely require a similar amount of code, but note that that those hooks need to be written only once by an experienced user and then distributed as part of Mallard.

Finally, since the console can display HTML output blocks, the user can write JavaScript code to implement widgets that control ML algorithm hyperparameters or thresholds. For instance, the user can create standard HTML5 input controls such as a range slider to tune a hyperparameter and then see how those adjustments affect algorithm performance.

Figure 6 shows an example widget from one of our case studies. The user has run a sentiment analysis [14] on their Twitter feed and created a slider to adjust a threshold. When they slide to a threshold value, all tweets with sentiment values below that value will be hidden. This enables users to customize their Twitter feed to hide the most negative-sounding tweets.

Step 4: Augment Host Webpage with Analysis Outputs

Mallard is unique in not only using webpages as data sources (Step 1) but also in using them as substrates to display the outputs of ML analyses in their original contexts. In contrast, when running a desktop or cloud ML workflow, the user sees only textual outputs on a command-line terminal or as output

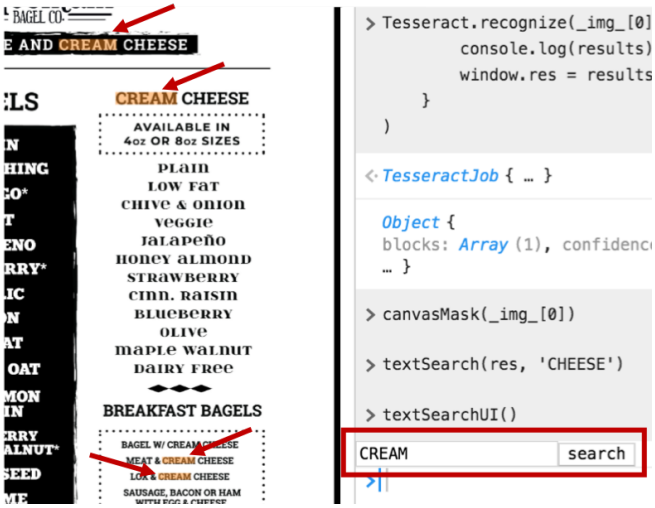


Figure 5: Creating a custom search widget (red box) in Mallard's console to perform OCR on an image in the host webpage, then using image overlay scaffolding to highlight the recognized text (Case Study 2).

data files that they need to manually sift through, which are both disconnected from the context of their input data.

When first acquiring each piece of data from the host webpage in Step 1, Mallard keeps track of the path to its DOM element. This way, the user can later augment that element with the outputs of their ML analyses. Here are details for each supported data type from Step 1:

- **Multimedia data such as images, videos, and audio clips**: Mallard can add an HTML caption below the original data's DOM element on the webpage to show relevant analysis results or replace it entirely with a new piece of data. For instance, it can replace an image on a webpage with a transformed version after applying a style transfer model (see Step 4 of Figure 2). Mallard can also use the image/video overlay scaffolding (see Figure 5) to augment images and videos on the host webpage.
- **Structured tabular data** in HTML tables: Mallard keeps a mapping between each piece of data in the parsed JavaScript data frame object and its original DOM element in the webpage's HTML table. This lets the user write code to, say, transform all data in a particular column and write their updated values back into the HTML table. Or they can insert an additional column in the HTML table to show additional analysis results alongside the original data points.
- **Unstructured plain text**: Mallard keeps track of what text was selected on the page to turn into a given JavaScript object. This provenance lets the user later transform that selected text by, say, altering its CSS with color or format updates. For instance, altering text color can indicate positive or negative tone in text sentiment analysis.
- **JavaScript objects**: Mallard enables users to acquire the value of a JavaScript object from any host webpage using the `:acquire` command. Likewise it also allows them to push those values into webpages that are opened in other tabs using a

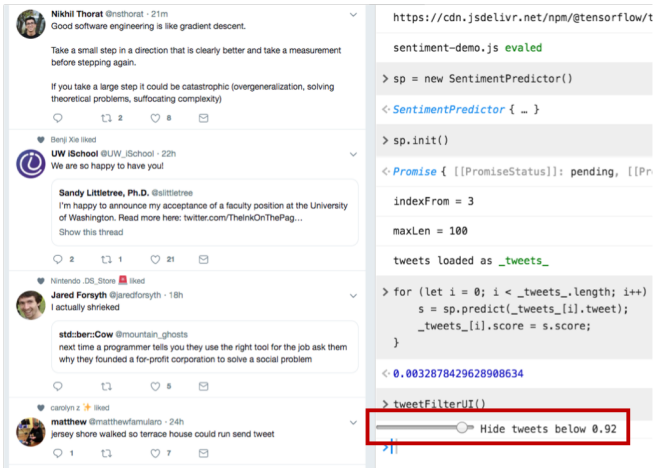


Figure 6: Creating a numerical slider and using it to hide tweets that fall below the selected threshold in a text sentiment analysis (Case Study 1).

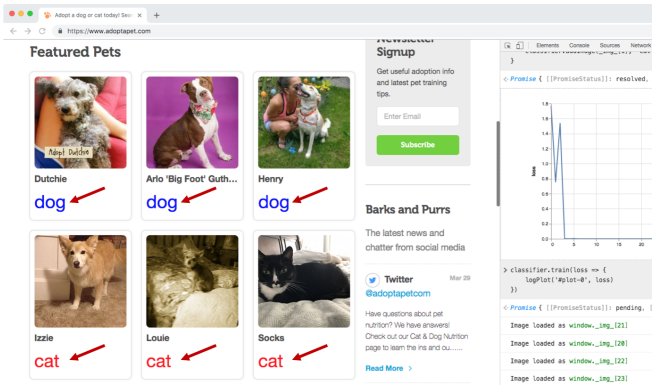


Figure 7: Testing a cat vs. dog image classifier and augmenting a pet table on a host webpage with the predicted class labels (Case Study 5).

`:push ${variable name} ${tab ID}` command that specifies both the variable name and the numerical ID of the other tab. This feature lets users prototype apps that span multiple webpages by forming a conduit (through the in-memory global storage within Chrome's developer tools) to pass JSON data between pages that may be hosted on different domains, without cross-domain restrictions.

Step 4 of Figure 2 shows an example of augmenting the Google Image Search HTML table with the results of running the style transfer ML model on each individual image. Also, Figure 7 shows a zoomed-in view from one of our case studies where we used Mallard to augment an HTML table on the host webpage with ML outputs. We first trained a cat vs. dog binary classifier and then applied it to all images on a pet adoption webpage, which augmented each pet's HTML table cell with its predicted class (cat or dog). This way, users can see model outputs within the context of the host webpage.

DISCUSSION: SYSTEM SCOPE AND LIMITATIONS

Mallard turns the web browser into a contextualized prototyping environment ("breadboard") for machine learning and related apps. Its technical contribution is a set of in-browser scaffolding to assist programmers throughout their workflow, summarized in Table 1. Beyond ML applications, Mallard can also be used to create browser extensions for general data science, personal data analytics, or web mashups. But doing so effectively would likely involve different forms of scaffolding, such as what Fusion provides for web mashups [73].

Just like prototype circuits constructed on breadboards, ML-based apps created with Mallard are *not* meant to scale or be deployed publicly. Although web browsers are now capable of handling more data and computation than their predecessors, they still cannot compete with dedicated servers in terms of running production-scale ML pipelines. Regarding security and privacy, Chrome extensions have extensive permissions to access private user data, so users should not install and run Mallard prototypes from untrusted sources.

Mallard is best suited to assist users in applying pretrained ML models on web data or to do some retraining (transfer learning [55]) by adding small to moderate amounts of their own data. Again due to scaling issues, it is impractical to use Mallard to train new models from scratch on large data sets or perform fundamental ML research such as creating new neural network architectures. However, we envision that as ML frameworks and pretrained model collections grow in maturity, there will be far more *users* of existing ML components (the target audience for Mallard) rather than developers of novel ML technologies.

Since each ML framework has its own APIs and data structures, certain parts of the scaffolding in Table 1 must be written as hooks into the APIs of each framework, marked with ⋆ in Table 1. As a proof of concept for our case studies, we implemented hooks for TensorFlow.js, one of the most popular ML frameworks [64]. (We also used ml5.js and face-api.js in our case studies, but those are built atop TensorFlow.js so they can reuse our same hooks.) Each hook took around a dozen lines of code and needs to be written only once by an expert; they can then be packaged and reused by many novices.

Mallard does not require users to install software or set up servers to host their data. But since Mallard relies on live webpages to host the data for ML prototypes, if those pages' contents change in the future, then analysis results could also change. Sometimes that is desirable, such as when one wants to re-run analyses on the latest fresh data on a webpage. But if a user really wants to snapshot the current moment's data, they can use the `:tojson ${variable name}` console command to serialize it as part of the session JSON state.

Mallard's scaffolding (Table 1) can lower barriers to ML prototyping in the browser by eliminating infrastructure setup and excessive boilerplate code. However, that scaffolding does not provide any more expressive power beyond writing regular JavaScript code in a Chrome extension.

Finally, Mallard eases some technical barriers to applying ML, but it does not help novices better understand either the mathematical or the ethical nuances surrounding complex ML models. One specific danger of more ML models and frameworks being available as opaque black-box components

Step 1: Acquire Data	
Multimedia import	parse images, videos, and audio clips
Table import	parse HTML tables and .csv/.tsv file links
`:acquire` command	import JavaScript object state into Mallard
`webcam()`, `mic()`	record from webcam or microphone
Step 2: Prototype	
`:load` command	remotely load npm or JS modules from web
`:tojson` command	serialize object state to JSON
Step 3: Visualize	
`:output` command	create visualization or interactive widget
Image/video overlay	augment with HTML canvas overlay
Training set histogram	ML debugging (⋆10 lines of hook code)
Prediction barplots	ML debugging (⋆5 lines)
Confusion matrix	ML debugging (⋆37 lines)
ROC curves	ML debugging (⋆16 lines)
Accuracy/loss curves	ML debugging (⋆11 lines)
Step 4: Augment Host Webpage	
Multimedia augmentation	add image captions, replace image, image overlays
Table augmentation	replace table cells, add rows or columns
Text augmentation	color and style highlighted text
`:push` command	push JavaScript object to another browser tab

Table 1: Summary of the scaffolding that Mallard provides to help users prototype ML in the browser without writing as much boilerplate code. ⋆ means we had to write ML-framework-specific hooks for this part.

is that programmers may misuse them in ways that perpetuate algorithmic biases [22, 25, 26, 28, 43]. For instance, facial recognition has the potential to amplify discrimination and other unethical uses of surveillance [61, 67]. Thus, we recommend for tools such as Mallard to be used alongside learning resources that demonstrate responsible uses of ML.

CASE STUDY OF PROTOTYPING ML WITH MALLARD

What is the range of ML applications that can be feasibly prototyped with Mallard? How much coding effort do they require? What are the limits of Mallard's capabilities? As a first step toward answering these questions, we performed an informal case study by using Mallard to prototype nine ML applications spanning domains such as sentiment analysis, image recognition, and gaming. We came up with prototype ideas by browsing online ML tutorials to get a sense of what instructors, students, and hobbyists try when exploring popular frameworks. We first summarize our nine prototypes:

1. Augmenting Amazon and Twitter with Sentiment Analysis

We augmented the Amazon and Twitter websites with a sentiment analysis NLP model, which analyzes text to assess whether its tone is positive or negative. We did so by loading TensorFlow.js in the Mallard console, downloading Google's pretrained sentiment analysis CNN (convolutional neural network) from a URL [14], and iteratively tuning its settings on blocks of text extracted from those webpages. For Amazon, we browsed to customer reviews on a product page and dragged blocks of text into Mallard's console to test and tune the model. For Twitter, we went to our home page of tweets and dragged in the text of individual tweets into the console.

Once we were satisfied with our model's settings, we wrote custom JavaScript code to inject into each host page. For Amazon, our code parses all of the text in the reviews section, runs sentiment analysis on each review, and highlights each review with a shade of green or red depending on how positive or negative it appears, respectively (Figure 8-1). Figure 6 shows our Twitter augmentation, where we made a slider widget that the user can slide to choose a threshold value. Our code then hides all tweets in their feed whose sentiment scores are below that threshold. This way, users can browse Twitter without seeing overly negative content. This case study shows how people can use Mallard to import a pretrained ML model, test it on text from existing webpages, and then create custom interactions to augment those pages.

2. Augmenting Restaurant Menus with OCR-Based Search

Many local restaurant webpages post their menus as scanned images, which makes it infeasible for viewers to perform text searches on their contents (Figure 8-2). To enable search-based interactions, we loaded the Tesseract.js [16] OCR (optical character recognition) model in the Mallard console and used it to extract text from those menu images. We iteratively tested and tuned the model in the console with some test images, and once satisfied, created a search box widget. When the user types a string in that box, our code searches for that text and retrieves the bounding boxes of where they appear in the original menu image. It then uses the image overlay scaffolding to render an orange highlight over those parts of the image to visually show the search results.

3. Classifying Birds on Stock Photo Websites

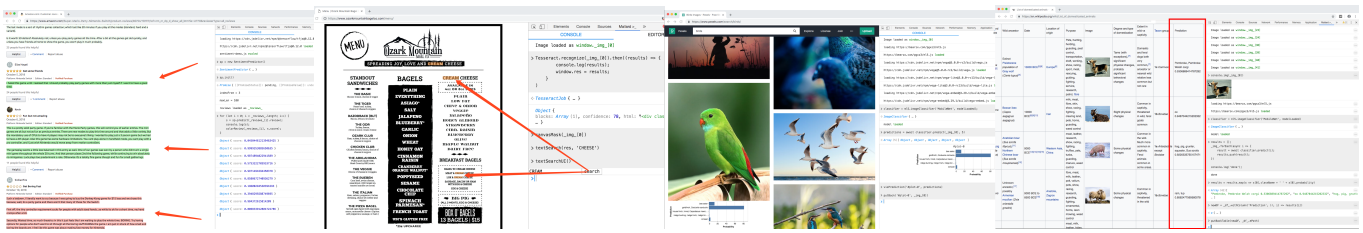
Stock photo websites offer large collections of professional photos, but they often lack meaningful labels. For instance, searching for birds on a photo site such as `pexels.com` will return many unlabeled photos. To add labels for bird species as image captions, we loaded up the ml5.js framework and MobileNet pretrained model [46] in Mallard. We dragged a few test bird images into the console to tune the code needed to classify them based on the corpus of thousands of animal species names stored in that model. Then we used Mallard's webpage augmentation scaffolding to augment the original image with a barplot showing the top predicted species names and their relative confidence scores (Figure 8-3).

4. Augmenting Wikipedia Tables with Animal Species Labels

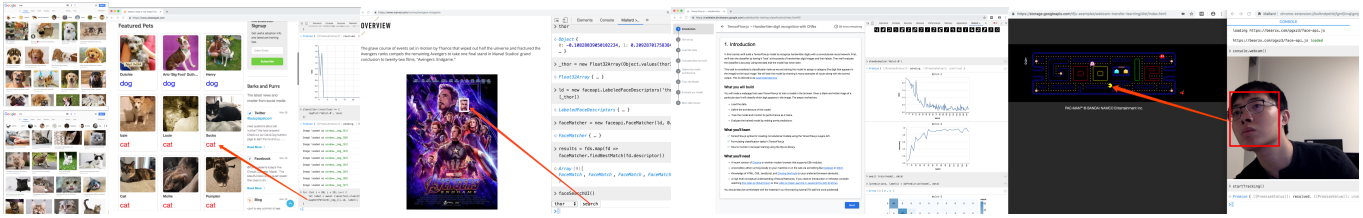
In a similar vein, Figure 8-4 shows how we used Mallard to augment a Wikipedia HTML table with animal species prediction results from a MobileNet model. We browsed to the "List of domesticated animals" Wikipedia page [7], which contains a table of animal images and descriptions. We dragged that table into Mallard, which parses it into a JavaScript data frame object with a column containing image objects. We ran the same MobileNet image classifier to get species names for all the animal images and used Mallard's table output scaffolding to augment the original HTML table with a new column showing its prediction results. This demo shows how Mallard can be applied to structured data such as tables to parse, analyze, and augment them with ML results within the context of the original page.

5. Retraining a Neural Network for Binary Classification

So far our case studies have only involved applying and tuning pretrained ML models, which is a reasonable starting point for hobbyists. A more advanced use case is performing transfer learning [55], which means retraining a neural



1. Amazon sentiment analysis 2. Restaurant menu OCR and search 3. Stock photo classification 4. Augmenting Wikipedia with ML



5. Retraining neural net on images 6. Training superhero recognizer 7. ML tutorials with live coding 8. Pac-Man with face tracking

Figure 8: Screenshots from our case studies where we used Mallard to augment webpages with ML model outputs and interactive controls. Case Study 9 (image style transfer) is shown in Figure 2.

network model by adding small to moderate amounts of additional data. Retraining takes significantly less data and time than training a new model from scratch since it can reuse most existing layers of a pretrained neural net such as MobileNet. For this case study we used retraining to build a custom binary classifier that differentiates between cats and dogs. To do so, we performed Google image searches for "cats" and "dogs" respectively, and dragged in 10 images of each into Mallard. We fed those images and labels into the MobileNet model and rendered the training loss curves using Mallard's ML debugging visualizations. By monitoring this curve as we added additional training data, we can see when the loss converges to a reasonable minimum. To test the model, we went to a pet adoption website [1] and augmented each image on there with the model's prediction labels (see Figure 7).

6. Training a Superhero Face Recognizer

As another example of retraining, using the base model from face-api.js [54] we trained a custom face recognizer to find Marvel Avengers superheroes. To do so, we first browsed to various webpages containing images of those superheroes, dragged them into Mallard, and added them to our labeled training set. After training completed, we built a selector UI in the Mallard console where the user can choose the name of an Avengers superhero and have the face recognizer try to find that superhero on a chosen image on the current webpage. For instance, they could choose a movie poster such as one for *Avengers: Endgame* [2], which contains the faces of over a dozen superheroes (Figure 8-6). If the model finds a match, it will highlight the bounding box of the superhero's face in that movie poster. This case study shows how Mallard lets users collect and aggregate training data from multiple webpages.

7. Augmenting ML Tutorial Webpage with Live Coding

ML tutorial webpages often contain sample code and data alongside their instructions. For instance, the TensorFlow.js home page has a step-by-step tutorial on how to train a numerical digit recognizer using the MNIST handwriting database [15]. To follow such a tutorial, normally the user would need to set up their own software development environment and load data sets into there. With Mallard, though, they can instead follow along with sample code directly on the tutorial webpage itself. To do so, we load TensorFlow.js and the MNIST data set from URLs and copy the tutorial's sample code into the Mallard console to do model training. Again we can monitor the loss curves in the console to see training performance. This case study shows how Mallard can turn static tutorial webpages into live coding playgrounds where learners can tinker with sample JavaScript code.

8. Playing Pac-Man Game using Real-Time Face Tracking

For a more interactive demonstration of Mallard's capabilities, we used it to build a face-based gaming controller for a Pac-Man game we found on the web. The idea is that by tracking the user's face in real time through a live webcam feed in the browser, that will enable them to control Pac-Man's movements by moving their head up, down, left, and right. To implement this custom controller, we loaded face-api.js and created a webcam block in Mallard's console using `console.webcam()`. We wrote code to continually extract still image frames from the webcam video feed and pass it into face-api.js to perform face detection. Then we took the position differences in the bounding boxes returned by face-api on consecutive frames and injected that signal into an existing Pac-Man web game to control Pac-Man. One could also imagine hooking Mallard to microphone inputs and using speech recognition models to provide a voice interface to Pac-Man. This case study shows how Mallard can be used to prototype real-time interactions built upon ML models.

9. Performing Style Transfer on Google Images Results

Finally, for a more artistic demo, we hooked up a neural network for image style transfer [42] to the Google image search results webpage. Style transfer is an ML technique where the

Case Study	Interaction Type	Training?	Augment Webpage?	Lines of code
1. Sentiment Analysis	use slider to filter or style text based on sentiment	No	Yes	† 67A, 77T
2. Restaurant OCR	search for text and highlight with overlay in OCR'ed images	No	Yes	17
3. Image Classification	none	No	Yes	10
4. Wikipedia Tables	none	No	Yes	15
5. Train Cat/Dog Classifier	none	Yes	Yes	16
6. Train Face Recognizer	select name and highlight recognized faces on image	Yes	Yes	15
7. Augment ML Tutorial	live coding and debugging while following tutorials	Yes	No	318⋆
8. Pac-Man Face Controller	control a game using face movements	No	No	45
9. Image Style Transfer	select artistic style to apply to images	No	Yes	20

Table 2: Summary of case studies showing interaction types, whether an ML model was trained on new data, whether host webpage was augmented, and lines of code to implement each. (†67 lines for Amazon, 77 lines for Twitter. ⋆most lines copy-pasted from sample code on the tutorial webpage.)

artistic style from one image, such as a van Gogh painting, is transferred to other images. We began prototyping by dragging one test image into the Mallard console to tune style transfer settings from the pretrained model that comes packaged with ml5.js [4]. After we were satisfied with how the output image looked by previewing it in the console, we built a simple selection UI where the user can choose between three pretrained styles from the *Scream*, *Rain Princess*, and *La Muse* paintings. When the user selects a style, our code applies that style to all images on the Google image search results page. This case study is shown in Figure 2.

Reflecting on Our Prototyping Experiences

Table 2 summarizes our case studies, which demonstrate that Mallard can be used to prototype ML apps that span a diverse range of application domains and interaction types. Most took around a dozen lines of JavaScript code to implement. Our code involved a mix of web front-end programming (e.g., creating HTML widgets and writing callbacks to make them interactive), API calls to Mallard's scaffolding functions (e.g., to augment elements on the host webpage), and API calls to ML framework code. In our experience, the most challenging part was working with the low-level APIs of ML frameworks such as TensorFlow.js. Note that making these frameworks easier to use is outside the scope of Mallard, whose goal is to provide contextual scaffolding for users to tinker with such frameworks within the browser. In sum, Mallard cannot make ML inherently "easier" but can help novices get started by removing extraneous software infrastructure barriers.

Note that all these case studies would have been feasible to implement without Mallard but would have required much more friction in setting up IDEs and web servers and then learning various data scraping and authentication APIs (e.g., for Twitter). Mallard reduces this friction by allowing users to start from a context where code infrastructure and data are already set up in their browser on webpages that they are familiar with. Its scaffolding code and enhanced JavaScript console also make it more convenient to acquire data, take user inputs, visualize ML models, and augment the host webpage.

Despite the convenience of in-browser prototyping, here are some frictions we encountered: When loading content such as images or videos, some websites block remote fetches from JavaScript front-end code. In our case studies we did not encounter these problems much, and we could choose different sites that did not have such restrictions. If this becomes an issue in the future, we will consider a proxy server approach similar to Fusion's [73] where the backend does the resource fetching. There were also inconsistencies in how the code for ML frameworks and pretrained models were hosted. Hopefully as better community standards form around package formats in the future, that will make it easier for programmers to mix and match components much like how package managers such as npm [8] already make it easier to manage JavaScript libraries. Finally, browser extensions run in a sandbox that communicates with host pages only via cumbersome text-based message passing. We would have liked finer-grained settings that both protect the host page's privacy while also sharing state across multiple pages and developer tools.

In these case studies we tried to emulate the scope of breadboard-style ML projects that might appeal to educators or hobbyists. But we have not yet performed a field deployment or formal usability study to get Mallard into the hands of real users. Thus, we cannot make claims about the system's learnability, discoverability, or overall usability. Looking forward, ideally we should deploy Mallard alongside properly designed instructional materials to provide both the technical and social contexts surrounding the given ML techniques.

CONCLUSION

Throughout the past decade, rapid advances in open-source machine learning frameworks and publicly-available libraries of pretrained models have made ML more accessible to programmers who are not ML specialists. In parallel, the rapid maturing of the web as a ubiquitous platform for running code and hosting data has made it well-positioned to become a compelling substrate for ML applications. This paper aims to coalesce these two complementary trends with Mallard, a browser extension that enables hobbyists to prototype ML directly within the context of existing webpages. More broadly, the web has long been integral to software prototyping; programmers often use it to *opportunistically forage* for technical information, code examples, and tutorials [30]. However, they must constantly context-switch between web browsers (where resources are foraged) and development tools (where software gets built). Mallard points toward a future where hobbyists can opportunistically prototype entirely within the browser, unifying all of their code, data, and environment.

ACKNOWLEDGMENTS

Thanks to Kathleen Tuite and Priyan Vaithilingam for feedback and to a Google Faculty Research Award for funding.

REFERENCES

[1] 2019. Adopt the Perfect Pet. https://www.adoptapet.com/. (2019). Accessed: 2019-04-02.

[2] 2019. Avengers: Endgame. https://www.marvel.com/movies/avengers-endgame. (2019). Accessed: 2019-04-02.

[3] 2019. AWS Cloud9: A cloud IDE for writing, running, and debugging code. https://aws.amazon.com/cloud9/. (2019). Accessed: 2019-04-02.

[4] 2019. Friendly Machine Learning for the Web. https://ml5js.org/. (2019). Accessed: 2019-04-02.

[5] 2019. Google Colaboratory. https://colab.research.google.com/. (2019). Accessed: 2019-04-02.

[6] 2019. Keras: The Python Deep Learning library. https://keras.io/. (2019). Accessed: 2019-04-02.

[7] 2019. List of domesticated animals. https://en.wikipedia.org/wiki/List_of_domesticated_animals. (2019). Accessed: 2019-04-02.

[8] 2019a. NPM. https://npmjs.com/. (2019). Accessed: 2019-04-02.

[9] 2019. Observable. https://observablehq.com/. (2019). Accessed: 2019-04-02.

[10] 2019. Project Jupyter. http://jupyter.org/. (2019). Accessed: 2019-04-02.

[11] 2019. React - A JavaScript library for building user interfaces. https://reactjs.org/. (2019). Accessed: 2019-04-02.

[12] 2019. Repl.it - The world's leading online coding platform. https://repl.it/. (2019). Accessed: 2019-04-02.

[13] 2019. TensorBoard: Visualizing Learning. https://www.tensorflow.org/guide/summaries_and_tensorboard. (2019). Accessed: 2019-04-02.

[14] 2019. TensorFlow.js Example: Sentiment Analysis. https://github.com/tensorflow/tfjs-examples/tree/master/sentiment. (2019). Accessed: 2019-04-02.

[15] 2019. TensorFlow.js — Handwritten digit recognition with CNNs. https://codelabs.developers.google.com/codelabs/tfjs-training-classfication/. (2019). Accessed: 2019-04-02.

[16] 2019. Tesseract.js - Pure Javascript Multilingual OCR. http://tesseract.projectnaptha.com/. (2019). Accessed: 2019-04-02.

[17] 2019b. This year in JavaScript: 2018 in review and npm's predictions for 2019. https://blog.npmjs.org/post/180868064080/this-year-in-javascript-2018-in-review-and-npms. (2019). Accessed: 2019-04-02.

[18] 2019. UNPKG. https://unpkg.com/. (2019). Accessed: 2019-04-02.

[19] 2019. Vega - A Visualization Grammar. https://vega.github.io/vega/. (2019). Accessed: 2019-04-02.

[20] 2019. What-If Tool. https://pair-code.github.io/what-if-tool/. (2019). Accessed: 2019-04-02.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[22] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. 2006. Learning User Interaction Models for Predicting Web Search Result Preferences. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '06)*. ACM, New York, NY, USA, 3–10. DOI: http://dx.doi.org/10.1145/1148170.1148175

[23] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier

Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR* abs/1605.02688 (2016). **http://arxiv.org/abs/1605.02688**

[24] Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 337–346. DOI: **http://dx.doi.org/10.1145/2702123.2702509**

[25] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine bias: There's software used across the country to predict future criminals. *ProPublica* 23 (2016).

[26] Richard Berk and Jordan Hyatt. 2015. Machine Learning Forecasts of Risk to Inform Sentencing Decisions. *Federal Sentencing Reporter* 27, 4 (2015), 222–228. DOI: **http://dx.doi.org/10.1525/fsr.2015.27.4.222**

[27] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST '05)*. ACM, New York, NY, USA, 163–172. DOI: **http://dx.doi.org/10.1145/1095034.1095062**

[28] Indranil Bose and Radha K. Mahapatra. 2001. Business data mining – a machine learning perspective. *Information & Management* 39, 3 (2001), 211 – 225. DOI:**http://dx.doi.org/https://doi.org/10.1016/S0378-7206(01)00091-X**

[29] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. DOI: **http://dx.doi.org/10.1109/TVCG.2011.185**

[30] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1589–1598. DOI: **http://dx.doi.org/10.1145/1518701.1518944**

[31] Jason Brownlee. 2019. A Gentle Introduction to Learning Curves for Diagnosing Machine Learning Model Performance. **https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/**. (2019). Accessed: 2019-04-02.

[32] Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: Exploring the Power of Tables on the Web. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 538–549. DOI: **http://dx.doi.org/10.14778/1453856.1453916**

[33] Carrie J. Cai and Philip J. Guo. 2019. Software Developers Learning Machine Learning: Motivations, Hurdles, and Desires. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '19)*.

[34] Bryan Chan, Leslie Wu, Justin Talbot, Mike Cammarano, and Pat Hanrahan. 2008. Vispedia: Interactive Visual Exploration of Wikipedia Data via Search-Based Integration. *IEEE Transactions on Visualization and Computer Graphics* 14, 6 (Nov. 2008), 1213–1220. DOI: **http://dx.doi.org/10.1109/TVCG.2008.178**

[35] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 963–975. DOI:**http://dx.doi.org/10.1145/3242587.3242661**

[36] Leon Chen. 2019. Keras.js. **https://transcranial.github.io/keras-js/**. (2019). Accessed: 2019-04-02.

[37] Brendan Colloran. 2019. Iodide: an experimental tool for scientific communication and exploration on the web. **https://hacks.mozilla.org/2019/03/iodide-an-experimental-tool-for-scientific-communicatiodide-for-scientific-communication-exploration-on-the-web/**. (2019). Accessed: 2019-04-02.

[38] Victor Dibia. 2019. Machine Learning In The Browser. **https://blog.fastforwardlabs.com/2019/02/28/machine-learning-in-the-browser.html**. (2019). Accessed: 2019-04-02.

[39] Daniel Drew, Julie L. Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. 2016. The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 677–686. DOI: **http://dx.doi.org/10.1145/2984511.2984566**

[40] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542, 7639 (2017), 115.

[41] Jerry Alan Fails and Dan R. Olsen, Jr. 2003. Interactive Machine Learning. In *Proceedings of the 8th International Conference on Intelligent User Interfaces (IUI '03)*. ACM, New York, NY, USA, 39–45. DOI: http://dx.doi.org/10.1145/604045.604056

[42] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2016. Image Style Transfer Using Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2414–2423. DOI: http://dx.doi.org/10.1109/CVPR.2016.265

[43] Nina Grgic-Hlaca, Elissa M. Redmiles, Krishna P. Gummadi, and Adrian Weller. 2018. Human Perceptions of Fairness in Algorithmic Decision Making: A Case Study of Criminal Risk Prediction. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 903–912. DOI: http://dx.doi.org/10.1145/3178876.3186138

[44] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. DOI:http://dx.doi.org/10.1145/1656274.1656278

[45] James A. Hanley and Barbara J. McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[46] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). http://arxiv.org/abs/1704.04861

[47] David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 125–134. DOI: http://dx.doi.org/10.1145/1166253.1166274

[48] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. DOI: http://dx.doi.org/10.1145/2647868.2654889

[49] Josua Krause, Adam Perer, and Kenney Ng. 2016. Interacting with Predictions: Visual Inspection of Black-box Machine Learning Models. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 5686–5697. DOI: http://dx.doi.org/10.1145/2858036.2858529

[50] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI: http://dx.doi.org/10.1145/1502650.1502667

[51] Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. 2007. Eyepatch: Prototyping Camera-based Interaction Through Examples. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 33–42. DOI: http://dx.doi.org/10.1145/1294211.1294219

[52] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241. http://dl.acm.org/citation.cfm?id=2946645.2946679

[53] Adam S Miner, Arnold Milstein, Stephen Schueller, Roshini Hegde, Christina Mangurian, and Eleni Linos. 2016. Smartphone-based conversational agents and responses to questions about mental health, interpersonal violence, and physical health. *JAMA internal medicine* 176, 5 (2016), 619–625.

[54] Vincent Mühler. 2019. face-api.js: JavaScript API for face detection and face recognition in the browser and nodejs with tensorflow.js. https://github.com/justadudewhohacks/face-api.js. (2019). Accessed: 2019-04-02.

[55] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (Oct 2010), 1345–1359. DOI:http://dx.doi.org/10.1109/TKDE.2009.191

[56] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[57] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. 2010. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, New York, NY, USA, 37–46. DOI: http://dx.doi.org/10.1145/1866029.1866038

[58] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. **http://dl.acm.org/citation.cfm?id=1953048.2078195**

[59] Mark Pilgrim. 2005. *Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox (Hacks)*. O'Reilly Media, Inc.

[60] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788. DOI: **http://dx.doi.org/10.1109/CVPR.2016.91**

[61] Oscar Schwartz. 2019. Don't look now: why you should be worried about machines reading your emotions. **https://www.theguardian.com/technology/2019/mar/06/facial-recognition-software-emotional-science**. (2019). Accessed: 2019-04-02.

[62] Remy Sharp. 2019. Web based console - for presentations and workshops. **https://github.com/remy/jsconsole/**. (2019). Accessed: 2019-04-02.

[63] Daniel Shiffman. 2018. ml5: Friendly Open Source Machine Learning Library for the Web. *ADJACENT* (2018). Issue 3.

[64] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viégas, and Martin Wattenberg. 2019. TensorFlow.js: Machine Learning for the Web and Beyond. *CoRR* abs/1901.05350 (2019). **http://arxiv.org/abs/1901.05350**

[65] Chris Urmson and William "Red" Whittaker. 2008. Self-Driving Cars and the Urban Challenge. *IEEE Intelligent Systems* 23, 2 (March 2008), 66–68. DOI: **http://dx.doi.org/10.1109/MIS.2008.34**

[66] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. 2016. WaveNet: A Generative Model for Raw Audio. *CoRR* abs/1609.03499 (2016). **http://arxiv.org/abs/1609.03499**

[67] James Vincent. 2019. Gender and racial bias found in Amazon's facial recognition technology (again). **https://www.theverge.com/2019/1/25/18197137/amazon-rekognition-facial-recognition-bias-race-gender**. (2019). Accessed: 2019-04-02.

[68] Chiuan Wang, Hsuan-Ming Yeh, Bryan Wang, Te-Yen Wu, Hsin-Ruey Tsai, Rong-Hao Liang, Yi-Ping Hung, and Mike Y. Chen. 2016. CircuitStack: Supporting Rapid Prototyping and Evolution of Electronic Circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 687–695. DOI: **http://dx.doi.org/10.1145/2984511.2984527**

[69] Greg Wilson. 2006. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science Engineering* 8, 6 (Nov 2006), 66–69. DOI: **http://dx.doi.org/10.1109/MCSE.2006.122**

[70] Jeffrey Wong and Jason I. Hong. 2007. Making Mashups with Marmite: Towards End-user Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 1435–1444. DOI: **http://dx.doi.org/10.1145/1240624.1240842**

[71] Jiawei Zhang, Yang Wang, Piero Molino, Lezhi Li, and David S. Ebert. 2019. Manifold: A Model-Agnostic Framework for Interpretation and Diagnosis of Machine Learning Models. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (Jan 2019), 364–373. DOI: **http://dx.doi.org/10.1109/TVCG.2018.2864499**

[72] Xiong Zhang and Philip J. Guo. 2017. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 691–702. DOI: **http://dx.doi.org/10.1145/3126594.3126663**

[73] Xiong Zhang and Philip J. Guo. 2018. Fusion: Opportunistic Web Prototyping with UI Mashups. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 951–962. DOI: **http://dx.doi.org/10.1145/3242587.3242632**