mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks

Mary Beth Kery
Carnegie Mellon University
mkery@cs.cmu.edu

Donghao Ren
Apple Inc.
donghao@apple.com

Fred Hohman
Georgia Institute of
Technology
fredhohman@gatech.edu

Dominik Moritz
Apple Inc.
domoritz@apple.com

Kanit Wongsuphasawat
Apple Inc.
kanitw@apple.com
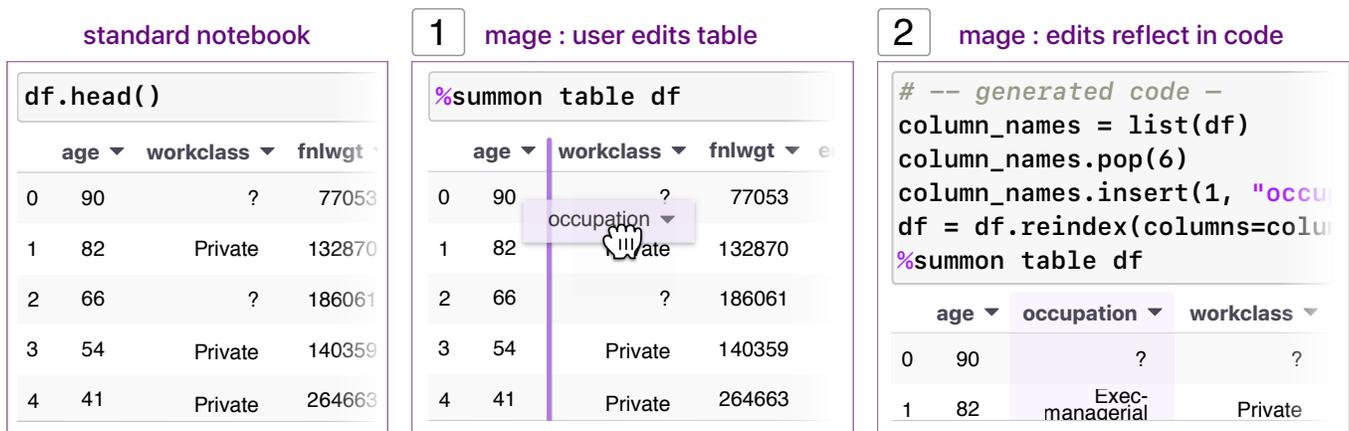
Kayur Patel
Apple Inc
kayur@apple.com

Figure 1: In a standard computational notebook, running code to refer to a data table `df` outputs a view-only rendering of the table. In a notebook powered by mage, the table is an interactive interface that the user can manipulate (1). When a user repositions a column in the interface (1), mage also automatically updates the code to reflect the change caused by the interaction (2).

ABSTRACT

We aim to increase the flexibility at which a data worker can choose the right tool for the job, regardless of whether the tool is a code library or an interactive graphical user interface (GUI). To achieve this flexibility, we extend computational notebooks with a new API mage, which supports tools that can represent themselves as both code and GUI as needed. We discuss the design of mage as well as design opportunities in the space of flexible code/GUI tools for data work. To understand tooling needs, we conduct a study with nine professional practitioners and elicit their feedback on mage and potential areas for flexible code/GUI tooling. We then implement six client tools for mage that illustrate the main themes of our study findings. Finally, we discuss open challenges in providing flexible code/GUI interactions for data workers.

Author Keywords

Data Science Programming; Machine Learning Programming; Handoff; Computational Notebooks;

CCS Concepts

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

INTRODUCTION

Data work is a complex domain where practitioners coming from diverse areas of expertise rely on a mix of tools to conduct their work [8, 13, 14]. Although code is the defacto power tool of choice for many professional data workers, graphical user interface (GUI) tools have an important role as well: charting tools, dashboards, and spreadsheets are all examples of pervasive forms of GUI tools for data work [8, 13, 14]. Graphical user interactions can provide easier inroads to novices, providing benefits such as recognition over recall [12], and are simply more practical for certain tasks, such as labeling image

data. In this paper, we aim to create an analysis environment where data workers have the flexibility to combine the complementary strengths of GUI and code and choose "the right tool for the job".

Computational notebooks make an ideal testbed for our research, as they are already set up for the purpose of blending mixed media with code. A notebook is designed as a series of "cells" that form a single document: cells of code, cells of text, and cells of output of various formats [17]. This interactive document format has made notebooks highly popular. Notebooks have been lauded by the scientific programming community for introducing new standards for replicability and understanding of data work — because explanations, code, and output can be read and executed together in one place [17, 20, 21, 25]. However, aside from standard application menus and panes that live in the periphery of a user's notebook document, today's notebooks put strong restrictions on the kinds of GUI widgets supported within documents. There is no such thing as a "GUI" cell. Today's community-created widgets are displayed in output cells, and include a variety of sliders, buttons, or visualizations that provide some, but very limited, forms of work. More deeply blending programming with GUI-based work in notebooks presents technical and design challenges, but, we argue, is a natural area of growth for notebooks towards supporting full workflows in one place.

It should be noted that many recent examples from research and practitioner communities point to a desire to move notebooks and data computation in blended GUI/code directions. Tools such as Wrangler [15], Wrex [6], Chartio, bamboolib, and qgrid[1] all focus on providing flexible GUI or code editing for data transformation in tables. As data tables is the canonical use case for so many of these tools, we demonstrate its appeal in Figure 1. In Figure 1 , the user sees a standard notebook computation where code *output* is a preview of a data table. As with most standard program output, notebook output is static and final, so the user is allowed to edit their table *within code only*. Alternatively, in Figure 1.1 suppose the notebook *did* offer GUI work in output. The user sees an interactive data sheet widget, where they can perform direct manipulation interactions as they would in spreadsheet software. Simply putting an interactive data sheet into the notebook is an imperfect solution, however, since data sheets don't produce a visible record of actions like code does and for this reason are notoriously prone to user mistakes [22]. This limitation can be overcome by combining GUI with code: the user edits the data sheet widget via direct manipulation while their actions are *simultaneously* reflected into auto-generated code (Figure 1.2), providing a repeatable record. Combining both GUI and code modalities gives users the complementary strengths of each as well as *choice*. The user is free to write code directly for some tasks or use direct manipulation to generate code for other tasks, as is most comfortable to them.

Our goal in this research is to design a generalized way to support a flexible GUI/code paradigm into a broad new class of data work tools. Our contributions are:

1. *mage*: A generalized application programming interface (API) **mage** that can power tools to move fluidly between code and GUI in a notebook. We discuss the design needs and tradeoffs for the API as well as how tool builders can integrate their work with mage.

2. *Demonstrating Design Space in Tools*: We implement six example notebook widgets to support data workers on tasks from machine learning to visualization support, powered by mage. We use these tools to illustrate key themes in the kinds of toolings that flexible GUI/code ability enables.

3. *Practitioners Reflecting on the Design Space*: To broaden our understanding of a flexible GUI/code tooling design space, we conduct an exploratory study with nine professional data works to gain their feedback and needs for the kinds of GUI/code tooling that might assist their daily work.

Next, we discuss the API design of mage as grounded in prior work and needs specific to a general-use API.

PRIOR WORK & DESIGN CONSTRAINTS

Aside from data table transformation work mentioned [6, 15], prior work in flexible GUI/code work systems has focused on visual domains, such as drawing [11], UI design [10], data visualizations [1], and commercial 3D modeling software. As these systems use various strategies for combining GUI with code, here we outline key design dimensions where mage draws upon, improves, or departs in key ways from prior work.

Allowing a user to make edits via GUI or code interchangeably in a live environment requires additional coordination in the underlying system. As the user works, a system must be able to keep both code and GUI synchronized based on a shared underlying state. Take for example a drawing app: if the user drew a star using a pen widget, the code for that drawing should update accordingly to represent a star. If the user then wrote code to make the star blue, the star and widgets on screen should update accordingly to show the blue star. Synchronizing between GUI actions and code edits becomes more complex the more power we give users to edit GUI and code interchangeably. Why? User actions in GUIs can have ambiguous or multiple possible translations to code. For instance, if a user pastes in a rotated star, is the star rotated positive 200 degrees or negative 160 degrees? User action in code come with the risk of there not being a GUI equivalent for that action. For instance, if the user writes mathematical constraints on their star shape, how should the GUI interface behave if it has no widget behavior built-in to represent constraints [11]?

For these reasons, common approaches carefully limit the expressiveness of what a user can do. One such approach, which is built-in for notebooks today[2], is *parameterization*. Parameterization allows users to designate variables in code that they can then change at runtime through sliders and other widgets. However, parameterization in the notebook today typically

---

[1]Spreadsheet widgets bamboolib or qgrid both generate some form of code within notebooks. Outside notebooks, Chartio has a visual table approach to forming SQL queries that also generates code.

[2]ipyWidgets **https://github.com/jupyter-widgets/ipywidgets**

only changes the runtime value of a variable and does *not* affect the value of the variable in written code[3]. This means that all GUI-based tuning only lasts for the current runtime session and is lost between sessions [10]. By generating output code based on user interaction, mage can help parameterization widgets address this pitfall and better match a key design ethos of notebooks to *support replicability* [17].

Beyond simple parameterization, another approach to balance between code expressivity and ambiguity is the use of a domain-specific language (DSL) where the creators carefully author mappings between a DSL and GUI actions [1, 15, 11]. Thus, for any action that the user takes, there is an equal representation of that action in both the code and GUI. However, every DSL is *by definition* designed for a specific task. Prime examples of DSLs in the data analysis space occur in commercial analysis platforms such SAS or JMP[4]. Both SAS and JMP are extensive GUI-based analysis tools fully compatible with code, but at the cost that all code must be in SAS's own or JMP's own bespoke DSL. As everyday data work involves in a myriad of tasks in a variety of programming languages, tying to just a single DSL conflicts with mage's goal as a general platform to enable a variety of tools. As a result, the mage API does not directly use any DSL, but instead allows *tool builders* to use DSLs in their own tools where appropriate. For example, our implementation of a plotting tool uses Vega-Lite [24] as the underlying representation for the charts.

Many systems have also used a programming by demonstration (PBD) approach to infer a correct mapping between what a user does in a GUI interface, and what the appropriate code to represent that should be [11, 6, 15, 9]. PBD has the benefit that, when carefully designed, it can ensure high-quality generated code synthesized fast enough to work in real time interactive applications [6, 9]. Again, however, we find the generality goals of mage's API are at odds with this approach. PBD, like domain specific languages, must be designed and tailored to a specific application, which is not feasible if mage must generalize to any data work application by any tool author. Again, a *tool builder* could use PBD where appropriate *internally* for their own tool, and their tool would be fully compatible to use with the mage API. The tension with generality is that the mage API itself cannot provide all the benefits of a PBD or DSL approach out-of-the-box, putting more effort on tool builders if they want to leverage these approaches.

As a departure from prior research, the goal of mage is not building a single highly flexible GUI/code tool, but rather creating a platform for an ecosystem of tools. Unlike a single closed system, like Sketch-n-Sketch for SVG editing [11], we cannot precisely model how a widget built with mage will affect state. Sketch-n-Sketch contributed a *complete and live 1-1 mapping* between GUI and code. This mapping meant that code updates in realtime to reflect actions within the GUI,

and conversely the GUI can update in realtime to reflect users' direct edits to the code. This flexibility to interact with either code or GUI in realtime is what we hope tools built with mage can enable. Next, we discuss how we design mage to achieve this goal, given the constraints that mage will not know precisely how any widget will behave.

MAGE SYSTEM OVERVIEW
The goal of mage is an API that any tool builder can use to make their GUI widget flexibly communicate back and forth with user code and state in a computational notebook. To facilitate this, mage is implemented as a Jupyter Notebook extension[5]. Since a Jupyter Notebook is itself a web app, mage is partly written in JavaScript to communicate with Jupyter APIs. As our prototype system focuses on Python notebooks, some elements of mage are written in Python.

Next, we use the `table` tool from Figure 1 as a use case to demonstrate how mage works, and how a tool creator can add bidirectional GUI/code communication to their widget:

Getting Started
`table`: Imagine the creator of `table` wants to make an interactive spreadsheet-like tool that can be used to transform data in the notebook. They start by designing and building a small web app for the table UI (we built `table`'s UI as a single-page React app[6]). The UI for the tool widget can be developed entirely outside the notebook at this phase, by using placeholder data (e.g. a cars sample dataset to display in `table`) for what will later be data coming from the notebook user.

mage: The mage API first requires configuration. A client tool provides its name, its parameter requirements for the data it would like to receive from the user, and its UI view as a JavaScript class. When a user calls a client tool by its name, mage can create a new instance of the client tool's base JavaScript class and pass in all the parameters from the user. Note that by asking tool creators to specify the exact number and types that their GUI tool will accept for parameters, mage is able to screen a request from the user and show users tool-specific error message if they've tried to call the client tool with data it cannot accept.
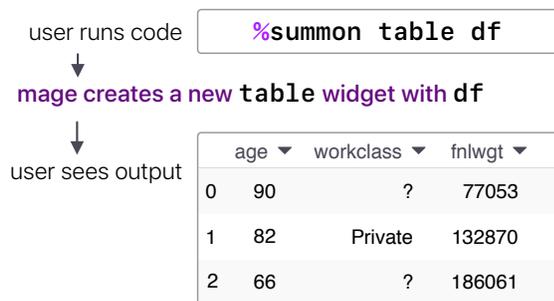
`table`: The tool creator registers "`table`" as their tool's name, and the parameter requirement that the user pass a variable type that can be displayed in a table, such as a pandas dataframe[7]. The tool creator also specified its base JavaScript class is `DataGrid` —though, what exactly will `DataGrid` do when it receives a Python binary object like a pandas dataframe? As an optional setting for situations like these, mage will accept a Python function, `preproccess`, which it runs on requested user data *before* passing the data to the client tool. The creator of `table` authors their `preprocess` function such that incoming data is converted into JSON objects that their system can read and display.

---

[3]Note that parameterization widgets *can change code* in some specific code editors, often for simple number or color variables. One implementations of notebooks that allows this is Carbide `https://alpha.trycarbide.com`

[4]SAS `https://www.sas.com/en_us/home.html` and JMP `https://www.jmp.com/en_us/home.html` are primarily GUI-based statistics and data analysis platforms that also support code in their own DSLs.

[5]Jupyter Notebook to date is the most popular open-source notebook platform, and offers extensive extension APIs `https://jupyter.org`

[6]Many of our example tools were created from a basic Create React App setup `https://create-react-app.dev`

[7]pandas is a common data manipulation library for Python optimized for performance with data tables `https://pandas.pydata.org`

mage: Now that mage has the configuration it needs, a user can call the `table` tool by running "`%summon table df`", where `df` is the user's own data variable. The figure below illustrates how the client tool appears for the user:

user runs code → `%summon table df`

↓

mage creates a new `table` widget with `df`

↓

user sees output

	age ▼	workclass ▼	fnlwgt ▼
0	90	?	77053
1	82	Private	132870
2	66	?	186061

To keep with existing notebook conventions, a client tool in mage is instantiated from code and displayed below in that code cell's output. mage leverages the ipywidgets library to render each tool's HTML div within a cell's output.

Users call mage via a custom *magics* command syntax `%summon <tool name> <parameters>`. Magics are a keyword used in iPython that start with `%`, and are conventionally used for meta-programming. We chose magics to highlight to a user that the output produced by this code behaves differently from normal notebook output, but a different implementation of this work could easily pick a non-magic syntax.

Choosing Which Actions Should Affect State

Now that a client tool can be called, displayed, and run in the notebook, a tool creator could choose to stop here. However, at this stage, even if the creator of `table` builds all manner of table manipulation functionality into their UI, their UI alone won't be able to affect the user's data `df` in any way. Crucially, Jupyter notebooks use *sandboxing* by design such that code that is not run by the user within their code cells cannot affect the user's program environment, and instead affects only a *copy* of the user's runtime. A user's variables cannot be directly modified without code run by the user.

Due to sandboxing, mage acts as the intermediary between a client tool and notebook state. For a client tool's data preprocessing steps mage simply runs these within a sandbox, since a client tool should not be changing the user's variables at this stage. Similarly, if a tool needs to request an updated value of a user's variable at runtime, mage provides a `getVariable` call that will read and return (but not modify) user variable values. For actions where the tool creator *does* want to modify state, we describe mage's approach next.

Templating Actions from GUI to Code

table: The creator of `table` decides that when a user filters a column in the table, as illustrated in Figure 2.1, they would like that filter applied to the user's data variable. To implement this behavior, when the user starts a filter in the UI, the `table` creator calls the mage API call `handoff()`.

mage: To avoid notebook sandboxing, it's best to run code from the user's own code cells. Thus, mage generates code needed to affect state and then injects it directly into the same cell where the client tool was originally instantiated. However, mage does not inherently "know" what it means to filter a column of data, nor does it "know" the meaning of any other user action. mage needs instructions on how to translate an action into code. To provide the flexibility and expressivity necessary to support a broad range of tools, we chose to use *code templates*. For each action that should affect state, a tool author provides templates to reflect that action in code.

table: The tool creator writes a template such that when a user filters a column (Figure 2.1), `table` has a fill-in-the-blank template corresponding to filtering a column using Python and pandas. The format of a template is illustrated in Figure 2.2, where names that start with "$" are blanks in the template and all other characters are literal pandas/Python syntax. When the action occurs, `table` calls the mage API `handoff(<template>, <data>)`, with the template as well as data it "knows" about the user's action. For instance, `table` "knows" based on its UI that the user is filtering column "age" with the value "age < 65", so both of these values are passed along with the template as data to help fill in its blanks.

mage: Upon receiving a `handoff()` API call, mage now resolves the template based on data the client tool provided as well as notebook state. For instance, the blank `$COL` in Figure 2.2 is filled in with "age", and the blank `$EXPR` is filled in with "< 65". Since a client tool is *not* responsible for knowing anything about the naming of variables in the user's environment, mage looks up the name of the user's data variable currently displayed in `table` and fills in the blank `$DF` with `df`. With all blanks in the template resolved, the code is complete Python syntax ready to run. An extra benefit of templates is that a client tool can send mage *multiple* templates for the same action, corresponding to the same action in different languages or library environments. In this way, a tool creator can support a broad range of programming environment with relatively little extra effort. If multiple template options are present, mage chooses the one that fits the current programming environment based on type checks.

Finally, mage inserts the new code into the code cell (Figure 2.3) and requests the notebook to re-execute the cell (Figure 2.4). Re-executing the cell has several effects:

1. Running the filter code ensures the user's variable `df` is now filtered and will remain filtered for all other subsequent code anywhere in the notebook that uses `df`. This filter is semi-permanent in that whenever the user starts a new session with their notebook or re-runs their notebook, the filter will execute on `df` each time. To remove the filter, the user may delete that code using the normal notebook undo.[8]

2. Running the cell refreshes the view of the tool `table` and delivers the updated (filtered) data for `df` to `table` through its `%summon` call parameter. The user sees a filtered table.

---

[8]Note that we did not implement mage undo support for this prototype of the API, but it should readily follow by translating back code to action. By having mage watch the notebook undo stack it can notifying a client tool if their most recent template has been undone.

	age ▾	workclass ▾	fnlwgt ▾
0	90	?	77053
1	82	Private	132870
2	66	?	186061

1 user selects filter by **age < 65**

2 **table** selects its filter template

```
$DF=$DF[$DF["$COL"]$EXPR]
```

table marks the user's choice

```
$DF=$DF[$DF["age"] < 65]
```

mage resolves the template as:

```
df = df[df["age"] < 65]
```

3 mage adds the newly created code to the original notebook cell and runs it to update state

4 user sees effect

```
df = df[df["age"] < 65]
%summon table df
```

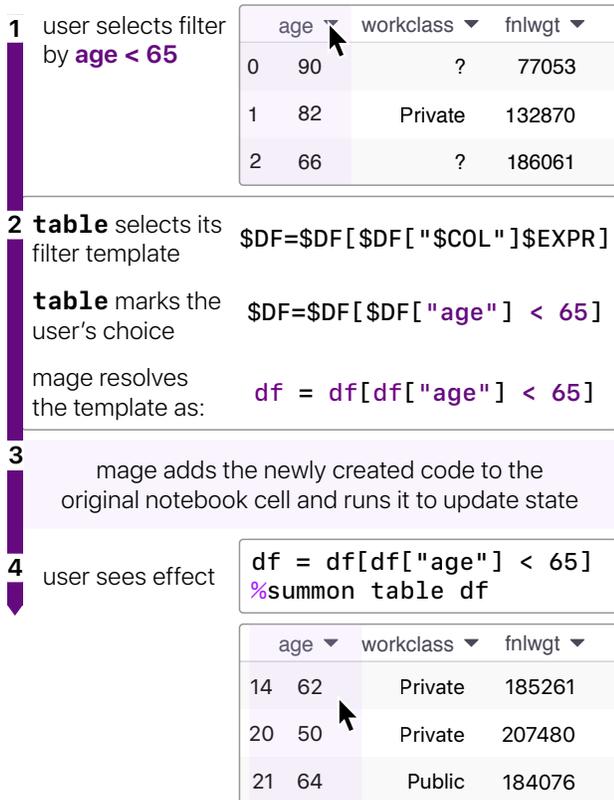|	age ▾	workclass ▾	fnlwgt ▾
14	62	Private	185261
20	50	Private	207480
21	64	Public	184076

Figure 2: Process for reflecting actions in code in mage

For each subsequent action, the tool's templates for that action will be turned to new code by the same process.

Reading Back Actions from Code to GUI
The goal of mage is to support *any* GUI tool in a notebook to work fluidly with code. However, many tools *will not have a complete mapping between GUI action and code*. To illustrate this challenge, we now switch our attention from `table` to a new client tool, an image editor shown in Figure 3. Like most GUIs, the image editor has a finite set of features: it supports common image processing tasks, but not every image processing task. Below, is the code that would be generated by mage if a user cropped their image with the `image` tool:

```
1   crop_img = dogs[1][0:750, 0:750]
2   %summon image crop_img
```

Now suppose the user makes some edits to the code directly:

```
1   crop_img = dogs[1][0:750, 100:850]
2   foo(crop_img)
3   %summon image crop_img
```

In line 1, the user has changed the dimensions of the image crop. mage watches code cells where a GUI tool is active and reacts to user code edits by attempting to update any state change back to the GUI tool. Instead of writing actions, here we use a client tool's code template to *read actions*: mage
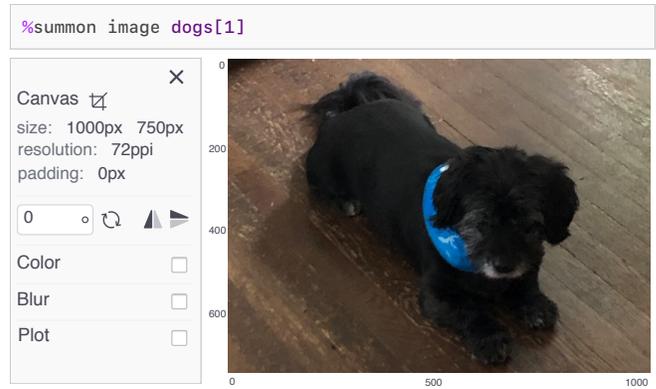

Figure 3: "image" is a basic image editor using mage. Unlike writing plain code for image processing steps, the user can instantly see the effect of their operation on the image.

takes templates from `image` and fills in template blanks to create a regex that will "recognize" any code matching that template. Here, the change to cropping is easily matched back to the crop action template. Finally, mage reports the updated crop action back to the tool `image`.

Note that regular expressions are less powerful than other ways of recognizing code, such as using an abstract syntax tree (AST) analysis. We choose to use regular expressions (in conjunction with type requirements provided by a template) as a conservative approach to limit recognizing false positives and to limit complications brought in by variables. For instance, a user in line 1 could write "n" instead of `850`, where the value of `n` is `850`. A regex will reject "n" for not being a valid number, while a more thorough static analysis could easily recognize "n" as a variable representing a number. However, recognizing `n` comes at the cost of mage needing to watch the value of `n` over time in case it changes from `850` in the future. Why? If `image` is not notified as soon as the crop value is no longer `850`, users may continue to use the image editor while seeing an invalid crop. We defer these more complex state handling cases to future work, and for now avoid many of them with conservative approaches.

The new code the user has added to line 2 represents a different scenario. The function `foo` is not within the `image` GUI tool's limited action template list and mage has no inkling about how `foo` affects the image `crop_img`. Here mage responds by running the code, such that `image` displays the correct value of `crop_img`, but with no "understanding" of any sequence of actions that precedes `foo`. Since we cannot know if `foo` depends on the order of operations of the crop in line 1, mage takes a conservative approach and assumes all code preceding an unrecognized line of code is untouchable user code. Thus `image` will have no undo operations possible for the image prior to `foo`, and all new actions will have auto-generated code inserted after `foo`.

GUI Widgets with mage vs. GUI Notebook Extensions
Due to the complexities of moving state between GUI/code representations, currently mage requires significant overhead

from tool creators, in the form of code templates and type information. A creator might worry if flexible GUI/code interactions is too high a burden to implement for their tool. To address this limitation, we look more broadly at tool creators' current workflows to find other avenues to lower overhead.

Today, aside from the simplest ipyWidgets which can be created by users on-the-fly, GUI widget tool creators for notebooks face much the same hurtles that tool creators face when extending any code editor. That is, creating a new widget is tightly tied to a specific editor's extension APIs. For instance, Jupyter Notebooks (used here for mage) has a different extension API from it's sibling notebook platform, Jupyter Lab. Given the still evolving nature of the notebook paradigm, today there are many notebook platforms to choose from, each with their own flavor of notebook programming and their own extension APIs. If we are to consider a GUI tools to be the same first-class means of work as code within a notebook, platform-dependency presents a major problem. A user can expect the same Python code to work the same across any notebook platform, but this will not hold true for GUI widgets.

Our aim in mage is to make a tool creator's GUI widget as generalizable as possible to different platform and language environments. Thus, a tool creator with mage does *not* take on platform-specific overhead, and provides only its own state-related logic: 1) an HTML-based GUI for their widget and 2) code templates to translate their GUI's interactions to user code. The mage API handles all notebook extension logic under-the-hood, which led to our implementation decision to build mage API on top of ipyWidgets API and the base notebook iPython runtime itself. The ipyWidgets API provides mage with some portability across the different notebook platforms that support ipyWidgets. Similarly, the base iPython runtime remains at the core of multiple different notebook platforms. Thus although mage API requires some modification to work in different notebook platforms, our core implementation is meant to be robust to different environments. Shielding individual GUI widgets from platform dependencies also comes with the benefit that changes to the underlying notebook extension APIs will only require update of the mage API and not individual implementation updates from tool creators.

DEMONSTRATING DESIGN SPACE IN TOOLS

At this point, we depart from discussing the mage API itself, to explore the design space of support tool interactions that a blended GUI/code environment powered by mage enables.

Thus far we have introduced two tools, `table` and `image` that leverage the mage API. Here, we describe all six tools, including the first two, that we implemented to explore different kinds of use cases for mage. To choose interesting use cases, our team first brainstormed common data or machine learning tasks. Next, we iteratively sketched possible widgets for each task, and then narrowed down into the final set of six tools that represent the most diverse set of interaction ideas to push our understanding of the design space. From a research perspective, actively developing client tools pushes on our understanding of GUI/code interactions by allowing us to actively play with them, and also pushes on the kinds of scenarios our mage API should deal with and support.
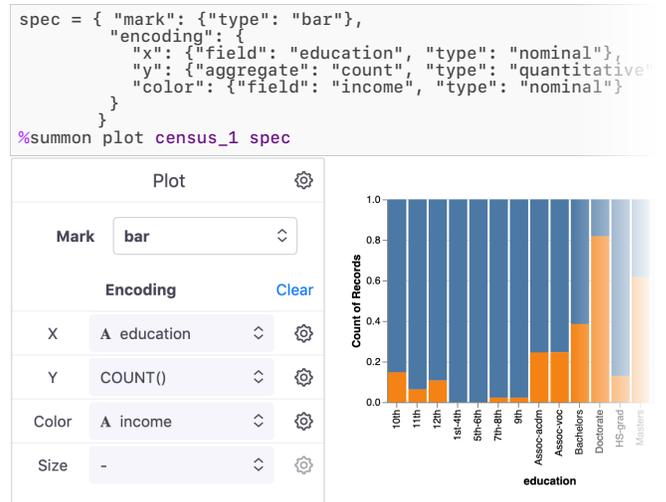


```
spec = { "mark": {"type": "bar"},
        "encoding": {
            "x": {"field": "education", "type": "nominal"},
            "y": {"aggregate": "count", "type": "quantitative"
            "color": {"field": "income", "type": "nominal"}
        }
    }
%summon plot census_1 spec
```

Figure 4: "plot" is a chart builder tool powered by Vega-Lite. Instead of specifying upfront what kind of chart they would like, a user inputs just their data into `plot` and can then try out a variety of different chart types and encodings in the GUI.

plot: **visualizing data as charts**

`plot`, shown in Figure 4, is a chart builder that aims to help data workers iterate on common chart designs without the need to recall an idiosyncratic chart syntax. `plot` uses a domain specific language (DSL), Vega-Lite [24], to represent state and to template its code. Akin to Voyager [28] and Tableau [26], users can use `plot`'s GUI to explore different visual encodings of their data. Using mage's API for live code generation, `plot` generates a Vega-Lite [24] specification based on user interaction. Although our current implementation of `plot` shows and reads Vega-Lite JSON specifications, future work can easily support bindings to Altair [27], the wrapper of Vega-Lite in Python.

As `plot` aims to help users build common charts, it is intentionally less expressive than Vega-Lite, which supports a broad variety of charts, and thus supports only a small subset of Vega-Lite features. One challenge with this expressivity mismatch is that the code and GUI state can be in conflict. Similar to prior code/GUI tool like Adobe Dreamweaver[9], `plot` avoids such conflicts by ensuring that every Vega-Lite attribute either has a widget for all of its possible values or have no interface for the attribute at all. Basically, every widget in the GUI must be able to display any change to its corresponding attribute in the code. For other Vega-Lite attributes without corresponding widgets, there are implicitly no conflicts as the attributes are not shown in the GUI.

table: **transforming and cleaning data**

`table`, as we have previously discussed and shown in Figure 1, is a data sheet tool for transforming data through direct manipulation, much like common spreadsheet tools but without spreadsheet formulas. The goal of `table` is to make data

[9]Adobe Dreamweaver blends code editing with GUI editing for website design https://www.adobe.com/products/dreamweaver.html
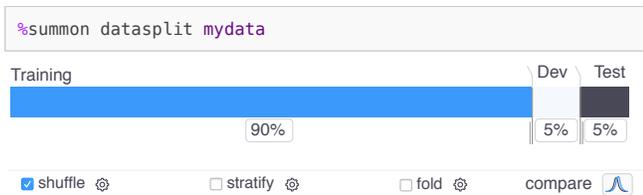
Figure 5: "datasplit" is a tool for a standard operation of splitting data into subsets for training and testing a ML model. `datasplit` gives the user operations to explore choices around creating data subsets. The color-segmented bar visually offers the users a suggested default split, as well as tunable divides that allow them to explore different splits.

transformation and cleaning easier for data workers to do and to visually verify that their data looks correct during this work. So called "data wrangling" can be the most tedious and time-consuming part of a data worker's workflow [15].

`image`: editing image data

`image`, as introduced earlier in Figure 3, is a basic image editor that uses mage to generate code for processing image. Since media data like images, sound, or video already have custom display widgets in a notebook today, here we explore the reasonable next steps of using mage to provide media editors standard to different kinds of media data. Although a data worker can certainly write code to process image data, the goal of `image` is to give data workers a faster feedback loop to "see" the effects of their operations.

`datasplit`: segmenting data for machine learning

With the `datasplit` tool, we explore the possibility of using mage to give a visual face to common data work API calls that contain lots of options. We designed `datasplit` as a visual version of the common scikit-learn[10] function `train_test_split()`. This means that `datasplit` uses mage to write a call to `train_test_split()`, while providing a GUI to give the user the ability to play and explore with all the parameter options for the API call. While `train_test_split()` is one specific API call, the task it represents is a universal one for supervised machine learning: given a dataset, split the dataset into separate smaller datasets that can be used to train and test the model. We note that many library calls, especially in machine learning or data processing, are highly tunable with a large set of optional parameters. For a data worker, especially a novice one, the meaning and appropriate use of all these options can be hard to grasp [4]. Thus, the goal of `datasplit` is to explore the idea of giving data workers more play-based interactions to understand options.

`confusion`: exploring classifier model performance

`confusion` is a tool, shown in Figure 6, with the goal of helping data workers understand machine learning model performance in their notebook. A confusion matrix is a common visualization for the performance of a classifier. The matrix

---

[10]scikit-learn is a common novice-friendly machine learning library in Python https://scikit-learn.org/stable/
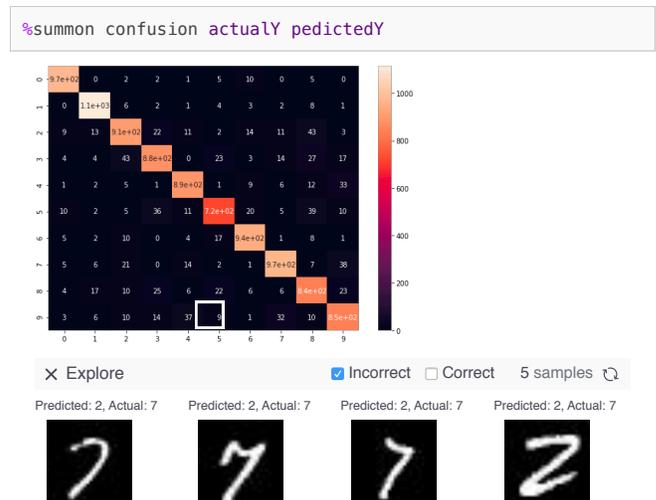

Figure 6: "confusion" is a tool to explore the correct and incorrect results of a classifier model, using a classic confusion matrix visualization. Unlike a plain confusion matrix, the user can interactively explore each result in the matrix, which appears in a row of samples beneath the matrix.

shows each possible label on the x and y axis: a perfect performance is one were all predicted labels fall in the left-to-right diagonal. Existing libraries like scikit-learn output a static table of numbers. By contrast, in a mage powered tool, the matrix becomes an interactive exploration interface where the user can click on individual matrix cells to see below examples of data that were (mis)classified in that way. As compared to tools we've discussed so far, an interesting property of `confusion` as an exploratory visualization is that its primary use of mage is not actually to generate code. Instead, `confusion` uses the mage API to retrieve and process data from the user's environment to automatically generate different ways for the user to look at their classifier results in the UI, without asking the user to provide any additional code.
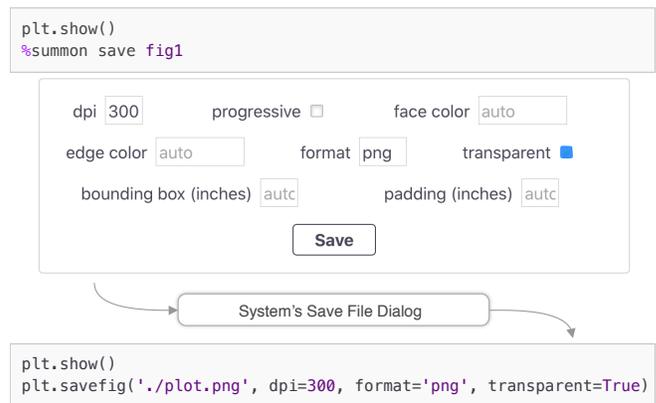

Figure 7: "save" is a tool, similar to a standard save file dialogue box, that lets a user interactively choose between the options for saving a chart or figure.

save: exporting (anything) to a file

Similar to `datasplit`, the `save` tool puts a visual GUI face to library calls for exporting data to a file. Shown in Figure 7, when the user runs `save` on a matplotlib plot, `save` presents a dialogue box containing the various options for exporting the plot to file. Once the user clicks the "save" button, they can then use a standard system file browser to save the file to their desired location, while `plot` generates appropriate saving code using mage. The key difference in idea for `save` compared to `datasplit`, is that we imagine `save` could be used as a single visual GUI utility for *many different* kinds of API calls that perform an export/save. Using mage to type-check the type of variable that a user inputs into the `%summon save <data>` call, `save` could contain bindings for different kinds of libraries, and adapt the dialogue box visualized accordingly. So, if the user inputs a pandas dataframe instead of a matplotlib plot, `save` will offer the save options offered by the pandas library accordingly.
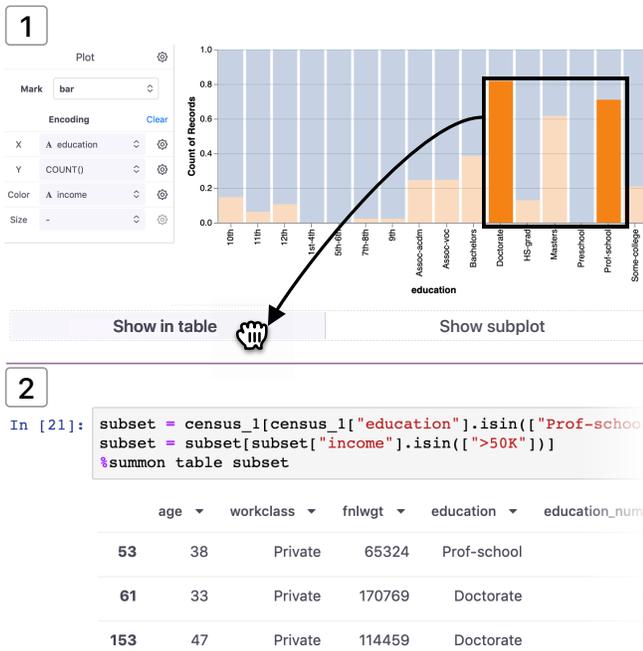

Figure 8: In (1) the user has the `plot` tool open and has constructed a chart. They select two columns from the chart of interest and begin dragging these columns outside of the chart. As soon as the user starts dragging, an option bin menu appears. The user drags and drops the columns to "Show in table", and in (2) a new cell is inserted immediately below that displays their selection both in code and in the `table` tool.

MULTI-TOOL USAGE SCENARIO

Now that we have demonstrated six example client tools for mage, an important design consideration is how a user might use their notebook with *multiple* flexible GUI/code tools working in concert. Although we largely leave the multi-tool design space for future work, we did design and implement one specific kind of interaction to illustrate the benefits fluid transitions between GUI/code tooling might offer users. We focused on one very fundamental data worker need: *selecting data*.

Widget Name	Common DS/ML Task
`table`	A spreadsheet widget that reflects direct manipulation actions in generated code.
`plot`	A chart that generates selection code when a user selects data through direct manipulation.
`image`	A basic image editor that generates code for common image data processing tasks.
`confusion`	An interactive tool for exploring classifier model performance.
`api`	A basic form interface for building correct calls to a web API to retrieve data.
`datasplit`	An interactive tool for splitting data into sets needed to train and test a model.
`save`	An interactive tool for saving a plot or figure with export settings.

Table 1: Initial paper prototypes shown to practitioners.

Imagine an analyst is exploring the 1994 US Census dataset from the UCI Machine Learning Repository [7] and wants to investigate historical gender bias in high income and high education workers. First, they call `%summon plot census` to visualize the distribution of education as a bar chart in the `plot` tool (Figure 8.1). The analyst then adds "income" to the color channel and enables "normalize" stacking for the "Y" axis to compare the percentage of low vs. high income in each education level (Figure 8.1). The analyst sees that higher education is correlated with higher income. Now, interested in the high education and high income group, they select the ">50k" income bars with "Doctorate" and "Prof school" education levels, and begin dragging this selection. As they drag the data selection out of the plot, mage provides a "drag bin" interaction where options specific to the `plot` client tool appear. The user sees two options for their selection: "Show in table" or "Show in subplot" (Figure 8.1). They drag their selection in to "Show in table", resulting in a new instance of the `table` tool appearing in a new cell below, with that data slice already loaded (Figure 8.2). In this new table view, the analyst finds that all five rows are "Male", leading to a hypothesis that there is a gender bias.

The interaction of dragging data selections between tools is the only multi-tool interaction currently built into the mage API. It can be used to drag a column from the `table` tool to visualize distributions with the `plot` tool or drag a matrix cell from the `confusion` tool to see the full `table` view of that particular data slice. Instead of GUI to GUI transitions, the user can also use drag to export a selection from a visualization like `plot` into a plain code selection for that data slice. Our goal is to make it simpler and more flexible for data workers to move the same data or data slices between different modalities.

PRACTITIONERS REFLECTING ON THE DESIGN SPACE

To better grasp practitioners' goals, interests, and values in this new design space of flexible GUI/code tooling, we recruited nine participants from a large technology company who both use notebooks and do professional data work. Most participants also worked with machine learning as part of their job. Each session began with a brief pitch for a new system for GUI/code work in notebooks that our team was in early stages of developing. We then asked participants to help our team

understand which tool and interaction ideas would be helpful (or not) for their own daily work.

To ground the discussion and help illustrate what we were talking about, we presented participants with seven starter tool ideas (listed in Table 1). Note that the `api` tool has the same basic idea as the `datasplit` tool in that they are both GUI tools to configure an API call, thus we did not actually implement the `api` tool.

Each tool was shown as a high-fidelity paper prototype, except for `table` and `plot`. These two were given as an initial example, chosen for their familiarity, and shown together within a live notebook. Participants were given the opportunity to try out the live `table` and `plot` tools with a simple data task to get a sense for the experience and feel of using a flexible GUI/code tool in the notebook. Each subsequent tool was then shown one at a time on paper to focus discussion.

While it may come as a surprise that we chose to present most tools on paper when it was fully possible to keep this design session in a live notebook with functioning tools, we made this choice to better enable conceptual feedback participants. Conventional user-centered design wisdom as well as prior design research [5] has found that when users are presented with high fidelity functioning prototypes, their feedback typically fixates on details of visuals and functionality of that particular implementation. In order to prevent distracting details from dominating the conversation and focus the user instead on considering the broader `concept` or ideas behind the tool, it is helpful to present less detail and lower fidelity for a tool to users [5]. Thus, we kept much of the discussion to single-screenshot paper prototypes of tools on paper.

When viewing each paper prototype, we asked participants to evaluate its "usefulness" as well as general impressions as if they were the one who would be using the tool. During this process, we encouraged participants to generate their own variants or entirely new tool ideas based on tasks they faced in their own daily work. We also encouraged participants to freely sketch or annotate directly on the paper prototypes.

Study Limitations

Note that by pushing participants to discuss more abstractly about concepts, rather than implementation details of mage or details of any of our demonstration tools, this study *does not* serve as a usability evaluation of mage. At this stage in research, we considered it higher-priority to expand our understanding of the core needs mage and flexible GUI/code tooling *should* support. An important next step of our future work is the evaluate the usability and efficacy of mage for serving these needs expressed by practitioners, which we detail next.

Results

Following the study, we conducted a thematic analysis of participants' tool ideas and feedback, which interactions they valued or rejected and why. We next discuss these themes and illustrate how these themes play out with respect to our six implemented example mage client tools.

Automatically Generating Code

Our nine participants held a mix of experience, as is common with data workers. Some practitioners had expertise in statistics or ML but were relatively new to programming, while some practitioners were expert programmers with less data/ML experience. More senior practitioners had expertise in both. A key theme we found was how expertise impacted how practitioners reacted to the idea of a flexible GUI/code tool automatically generating code for them.

Participants who were less confident with programming particularly appreciated the ability of mage tools to generate code for them while allowing them to use more familiar modalities like the data sheet `table`. Some more novice programmers even expressed a preference that generated code be hidden by default, so that they did not even have to look at it. Some notebook environments like Google's Colab[11] do offer code hiding features, where a code cell can be hidden by default but then shown if expanded. Meanwhile, programming experts tended to reject some tool ideas that they felt they could do much faster by writing code.

An example of participants' divide on code generation is the `save` tool for exporting chart figures to file (Figure 7). Some participants greatly appreciated this functionality as a way to prevent them from having to memorize finicky syntax or make silly mistakes saving their chart incorrectly. However, programming experts said they were unlikely to use a tool such as this when they already know the correct syntax.

Visual Data Selection

All participants wanted easier ways to select data and use direct manipulation to pull a selection into code. This kind of interaction is demonstrated above under "Multi-tool Use Case" (Figure 8). Practitioners value data selection through direct manipulation for multiple purposes, such as being able to interactively label data or interactively choose what data to explore more closely.

Interact to Explore Model Performance

Since most participants dealt with developing machine learning (ML) models as part of their work, participants were particularly enthusiastic about having interactive tools to work with to help them understand model performance. Even with the availability of tests and metrics, understanding model performance is a challenging exploratory task for practitioners, especially for ML novices [19, 29, 3]. Interactive tools can help users better understand model performance [18, 23], and a benefit of mage is that client tools can take in a user's model data and context, directly within their active code environment, and provide in-the-moment interactive support in situ without switching to an outside GUI tool. An example of this is demonstrated in Figure 6, with the `confusion` tool. While the `confusion` tool was rated highest among all nine participants, usually `confusion` prompted a discussion for how participants hoped for such interactive tools to be available for much more complex model analysis tasks. One simple tool idea suggested by a participant was to have an interactive way

---

[11]Google Colaboratory is a web-based notebook environment that allows live multi-user editing **https://colab.research.google.com**

to explore trade-offs in performance of their model by letting them interactively tune their preferred threshold for precision versus recall.

Make Good Practices Easier
Participants were particularly excited about the ability in `datasplit` to compare distributions, as this is a best-practice that is rarely provided by default. Although of course all functionality in `dataplit` is *possible* in raw code, a GUI makes suggested operations readily apparent to novice users, who may not yet have the ML know-how to follow best-practices without scaffolding. Data and ML programming APIs ideally encapsulate best-practices as the default capabilities they provide to users [2]. Interactive GUI tools also have more opportunity to nudge users towards best-practices when placed directly into the user's work environment.

FUTURE WORK
Next, we cover areas of future work towards making mage more powerful and flexible for data workers and tool creators:

Future Work: Generated Code Quality
During exploratory programming, data workers try out many possibilities that don't always contributing to the final result [16]. Simply generating new code for each new user action can unfortunately result in an excess of redundant and obsolete code. A simple example is shown below:

```
1  df.insert(14, "educ", dat)
2  column_names = list(df)
3  column_names.pop(14)
4  column_names.insert(2, "educ")
5  df = df.reindex(columns=column_names)
6  %summon table df
```

Here the user inserts a new column at one position, then repositions it. This sequence should be simplified to just:

```
1  df.insert(2, "educ", dat)
2  %summon table df
```

For a single tool, code quality is straightforward to optimize. For the `table` tool, we implemented `table`'s own internal state dictionary object, which is updated by each action in the `table`. Instead of adding a new line of code for each action, mage gives the option of fully rewriting the *writable area* of all auto-generated code (unrecognized user code is considered not writable) in the user's code cell. So for `table`, after each user action `table` uses its own state dictionary to output a *list of templates* corresponding just to the final current state of the table. This results in compact code as above.

Our challenge lies in *generalizing* this kind of optimization to any tool, rather than leaving it to a tool author to implement. We found that to create a compact internal representation of state, as in `table`, relies on both having some knowledge of how to operationalize state into a dictionary and order of operations, both of which are specific to the tool domain. Improving this process in an area for future work.

Future Work: Movement Between GUI Tools
Although we prototyped some ways that a user can move data between two GUI tools (see the section on Multi-tool Use Case above), many questions remain open about how a user might transfer content between two GUI tools, and not just between GUI and code. A simple use case for this may be to create a set of coordinated visualizations, like an impromptu data dashboard, that all update based on data selection from a table. Or, similarly, a data worker may want several GUI tools designed for providing model metrics to display metrics specific to different data slices within the model or different tradeoffs between precision and recall. The chief technical challenge in coordinating multiple GUI tools in such a way is designing an easy-to-use protocol by which a tool creator can specify what kinds of *other tools* and other data signals their tool can work with. This adds extra complexity for tool builders, which would need careful design to ensure the process was as lightweight for creators as possible.

More generally an overall challenge for mage, or any future toolkit like it, is to make the process for individual tool builders easier. Despite providing tool builders a bulk of analysis and code generation to enable fluid GUI/code work, our work demonstrates a number of complications where tool-specific understanding is currently needed from tool builders.

CONCLUSION
In this work, we have discussed mage, an API for allowing smooth transitions between GUI and code work in notebooks, as well as new kinds of tool avenues this GUI/code back-and-forth enables, demonstrated through six example tools. Our hope is that this work can enable and inspire new support tools to give data workers flexible interactive help at the moment as they need it. However, the core idea of flexible transitions between tools is much bigger than our work here, or computational notebooks. Seamless transitions between different devices in a user's environment or seamless transition between code and visuals in 3D graphics programs are both common examples of this broader technique. For data work, blending modalities of different kinds of tools practitioners use offers a new interaction space for richer interactions and more context-aware support tailored to a practitioners' specific workflow. We hope to see more HCI systems work in these intersections to support data workers in coming years.

ACKNOWLEDGEMENTS

REFERENCES
[1] Alex Bigelow, Steven Drucker, Danyel Fisher, and Miriah Meyer. 2016. Iterating between tools to create and edit visualizations. *IEEE Transactions on Visualization and Computer Graphics* (2016). DOI: http://dx.doi.org/10.1109/TVCG.2016.2598609

[2] Joshua Bloch. 2006. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN*

symposium on Object-oriented programming systems, languages, and applications.

[3] Ángel Cabrera, Will Epperson, Fred Hohman, Minsuk Kahng, Jamie Morgenstern, and Duen Horng Chau. 2019. FairVis: Visual Analytics for Discovering Intersectional Bias in Machine Learning. *IEEE Conference on Visual Analytics Science and Technology (VAST)* (2019). DOI: http://dx.doi.org/10.1109/VAST47406.2019.8986948

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

[5] Scott Davidoff, Min Kyung Lee, Anind K Dey, and John Zimmerman. 2007. Rapidly exploring application design through speed dating. In *International Conference on Ubiquitous Computing*. Springer.

[6] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. (????).

[7] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. (2017). http://archive.ics.uci.edu/ml

[8] Gregory Piatetsky for KDnuggets. 2018. Python eats away at R: Top Software for Analytics, Data Science, Machine Learning in 2018: Trends and Analysis. (2018). https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html.

[9] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* (2011).

[10] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*. ACM. DOI: http://dx.doi.org/10.1145/1449715.1449732

[11] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. DOI:http://dx.doi.org/10.1145/3332165.3347925

[12] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human-computer interaction* (1985). DOI: http://dx.doi.org/10.1207/s15327051hci0104_2

[13] Kaggle. 2017. Kaggle Machine Learning & Data Science Survey 2017. Dataset. (2017). Available from Kaggle https://www.kaggle.com/kaggle/kaggle-survey-2017.

[14] Kaggle. 2018. 2018 Kaggle ML & DS Survey. Dataset. (2018). Available from Kaggle https://www.kaggle.com/kaggle/kaggle-survey-2018.

[15] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. DOI: http://dx.doi.org/10.1145/1978942.1979444

[16] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/3025453.3025626

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press.

[18] 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*. ACM. DOI: http://dx.doi.org/10.1145/1866029.1866038

[19] Kayur Patel, James Fogarty, James A Landay, and Beverly Harrison. 2008. Investigating statistical machine learning as a tool for software development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. DOI: http://dx.doi.org/10.1145/1357054.1357160

[20] Roger D Peng. 2011. Reproducible research in computational science. *Science* (2011).

[21] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* (2018). DOI: http://dx.doi.org/10.1038/d41586-018-07196-1

[22] Stephen G Powell, Kenneth R Baker, and Barry Lawson. 2008. A critical review of the literature on spreadsheet errors. *Decision Support Systems* (2008).

[23] Donghao Ren, Saleema Amershi, Bongshin Lee, Jina Suh, and Jason D Williams. 2016. Squares: Supporting interactive performance analysis for multiclass classifiers. *IEEE transactions on visualization and computer graphics* (2016). DOI: http://dx.doi.org/10.1109/TVCG.2016.2598828

[24] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* (2016). DOI: http://dx.doi.org/10.1109/TVCG.2016.2599030

[25] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature News* (2014).

[26] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* (2002). `DOI:` `http://dx.doi.org/10.1145/1400214.1400234`

[27] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software* (2018). `DOI:` `http://dx.doi.org/10.21105/joss.01057`

[28] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2015. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE transactions on visualization and computer graphics* (2015). `DOI:` `http://dx.doi.org/10.1109/TVCG.2015.2467191`

[29] Qian Yang, Jina Suh, Nan-Chen Chen, and Gonzalo Ramos. 2018. Grounding interactive machine learning tool design in how non-experts actually build models. In *Proceedings of the 2018 Designing Interactive Systems Conference*. `DOI:` `http://dx.doi.org/10.1145/3196709.3196729`