EDITORS: Joao Comba, comba@inf.ufrgs.br
Kelly Gaither, kelly@tacc.utexas.edu

DEPARTMENT: VISUALIZATION CORNER

Interactive Data Visualization in Jupyter Notebooks

Jorge Piazentin Ono, Juliana Freire, Claudio T. Silva, *New York University, Brooklyn, NY, 11201, USA*

Interactive visualizations are at the core of the exploratory data analysis process, enabling users to directly manipulate and gain insights from data. In this article, we present three different ways in which interactive visualizations can be included in Jupyter Notebooks: 1) matplotlib callbacks; 2) visualization toolkits; and 3) embedding HTML visualizations. We hope that this article will help developers to select the best tools to build their interactive charts in Jupyter Notebooks.

Jupyter Notebooks are widely used for data exploration, enabling analysts to write documents that contain software code, computational output, formatted text, and data visualizations. In fact, this article is written entirely in a Jupyter Notebook, which can be run by the interested reader in order to interact with the visualizations and explore the source code in more detail. The notebook is available on Zenodo (https://zenodo.org/record/4444154). Visualization is an essential component of the data exploration process, and can be frequently found in Jupyter Notebooks. For example, a recent study of public GitHub repositories found that *matplotlib* was the second most imported package in the notebook environment.[1]

INTERACTIVE VISUALIZATION IN JUPYTER NOTEBOOKS

When datasets are too large or too complex, interactive visualization becomes a useful tool in an exploratory data analysis. Interactive visualizations can enable, among many others, the display of information at multiple levels of detail, the exploration of data using coordinated views, and the dynamic change of the charts to focus on the user's interests. While notebooks have traditionally been used with static visualizations, it is possible to embed sophisticated interactive visualizations and support advanced visual analysis as well.

In this article, we present three simple and powerful approaches in which data scientists can create interactive visualizations in Jupyter Notebooks: *matplotlib* callbacks, visualization toolkits, and custom HTML embedding. These approaches offer a number of benefits and drawbacks that need to be considered by the developer so that they can make an informed decision about their visualization task. By the end of this article, the reader will have a good understanding of the three methods, and will be able to select an implementation approach depending on the level of interaction, customization, and data flow desired.

Matplotlib Callbacks

The *matplotlib* library[2] is the most popular general purpose visualization package for Jupyter Notebooks.[1] This tool enables the creation of static, animated, and interactive visualizations, which can be rendered directly as the output of notebook cells. However, the available user interactions are limited: There is support for click and keypress events, but drag-and-drop, tooltips, and cross-filtering, frequently supported in visualization tools, are not directly provided. To expand the possible user interactions, *ipywidgets* can be used. *ipywidgets* is a library that provides HTML form inputs in the Jupyter interface, including drop down menus, text boxes, and sliders.

Visualization Toolkits

In order to enable the creation of more interactive visualizations in Python and Jupyter Notebooks, many open-source visualization toolkits have been

1521-9615 © 2021 IEEE
Digital Object Identifier 10.1109/MCSE.2021.3052619
Date of current version 25 March 2021.

TABLE 1. Summary of interactive visualization approaches in Jupyter Notebook.

Library	Interaction	Output	Customization	Dashboard	Data Flow
matplotlib	Low	Flexible	Low	Limited	Bidirectional
Plotly	High	HTML	Low	Yes	Bidirectional
Bokeh	High	HTML	Low	Yes	Bidirectional
Altair	High	HTML	Low	Yes	Python → JavaScript
HTML Embedding	High	HTML	High	Yes	Bidirectional

developed. Among those, Perkel *et al.*[3] highlight *Plotly*, *Bokeh*, and *Altair*. These libraries are built on top of web technologies, and create visualizations that can be seen in web browsers. Syntaxwise, *Plotly* and *Bokeh* are very similar to *matplotlib*. However, both libraries have been developed with a focus on user interaction, enabling the creation of web-based dashboards that combine interactive widgets and charts, and support multiple user inputs, including click, drag-and-drop, tooltips, selection, crossfilter, and bidirectional communication with Python via callbacks. *Altair*[4] differs from the aforementioned libraries in the way visualizations are defined: It uses a declarative specification that ports VEGA-Lite,[5] a data visualization grammar, to Python. A wide range of interactive visualizations can be expressed using a small number of Altair primitives, making this library very flexible. However, the produced visualizations cannot communicate with Python, and therefore, the results of user interactions cannot be used in further computations.

Custom HTML Embedding

There might be cases when a visualization cannot be created using any off-the-shelf Python libraries. When this happens, the developer has the option to code the visualization using a web framework and embed it in the notebook. This option offers the most flexibility, as the visualization can be fully customized and interactions can be scripted on demand. JavaScript libraries such as React and D3 can be used to facilitate the implementation of custom visualizations.

Table 1 summarizes the different approaches to add interactive visualizations in Jupyter Notebooks. The approaches are classified in terms of interaction, type of output, level of customization, support for dashboards, and data flow. When creating a new visualization, we believe these properties should be taken into consideration.

INTERACTIVE VISUALIZATIONS IN ACTION

In this section, we will show how to create interactive visualizations in Jupyter Notebooks using three approaches discussed in the previous section: *matplotlib* charts, *Altair* specifications, and custom HTML visualizations. Since the syntax of *Plotly* and *Bokeh* are very similar to *matplotlib*, we will not cover them in this article. We refer the interested reader to their online documentations.

Matplotlib With Callbacks

In order to enable interactive *matplotlib* charts in the notebook environment, users need to activate this option using the *"%matplotlib notebook"* magic command (https://matplotlib.org/3.3.3/users/interactive.html). The produced charts will natively support pan and zoom operations, but can be configured to receive other types of user input, such as mouse click and key press, which can trigger the run of user-defined callback functions (https://matplotlib.org/3.3.3/users/event_handling.html).

After a chart is created, for example, using *pyplot.scatter*, the user events can be captured by setting callback functions on the *canvas* using the method *mpl_connect*. Multiple events are available, including *button_press_event*, *button_release_event*, *key_press_event*, and *key_release_event*.

We show a minimal example below, where the visualization draws points on top of the user clicks. The resulting visualization in shown in Figure 1.

This approach can add *click* interactions to a chart with a few lines of code. However, we are limited to the

```
[ ]: %matplotlib notebook
import matplotlib.pyplot as plt

fig, ax = plt.subplots(); # Creating an empty chart
plt.xlim([0, 10]); plt.ylim([0, 10]) # Setting X and Y axis limits
def onclick(event): # Callback function
    ax.scatter(event.xdata, event.ydata, color='steelblue') # Draw a point on␣
 ↪top of the user click position.

cid = fig.canvas.mpl_connect('button_press_event', onclick) # Callback setup
```

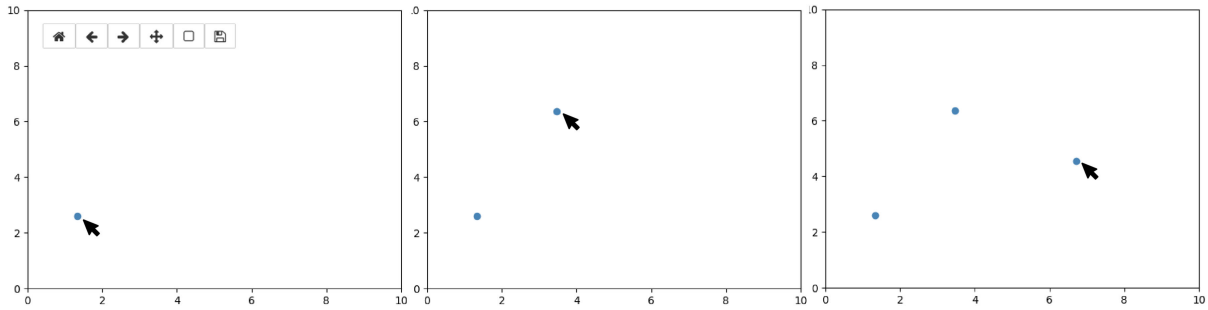

FIGURE 1. Interactive Matplotlib chart, where the user can click on the canvas in order to add a point at that position. The interactive chart also enables pan and zoom operations by default.

types of charts and interactions supported by matplotlib. When these options are not enough, the developer might need to consider other libraries, such as *Altair*, or creating their own visualization in HTML/Javascript.

Altair Specification

Altair enables the creation of interactive visualizations by using a pythonic port of the Vega-Lite specification.[4] Altair uses a declarative visualization paradigm: Instead of telling the library every step of how to draw a chart, the programmer specifies the data and the visual encodings, and the library takes care of the rest.

```
[ ]: import altair as alt
     from vega_datasets import data

     df = data.iris()

     alt.Chart(df).mark_circle().encode(
         x='petalLength',
         y='petalWidth',
         color='species',
         tooltip=['sepalLength', 'sepalWidth', 'petalLength', 'petalWidth', 'species']
     ).interactive()
```

In order to create a chart, the developer needs to have a *Pandas DataFrame* containing the data to be visualized. An *Altair.Chart* object needs to be created, with the corresponding *DataFrame* passed as a parameter. Next, an *encoding* and a *mark* need to be selected. *Encodings* tell *Altair* how the *DataFrame* columns should be mapped to visual attributes. Meanwhile, *marks* specify how the attributes should be represented on the plot (for example, as a circle, line, area chart, etc.).

We show a basic example of an Altair scatter plot with the Iris dataset (see Figure 2). The dataset contains information regarding 150 Iris flowers, with measurements of length and width of the plant, as well as the flower species. Data points can be hovered to show additional information as a tooltip (notice that this was not possible in *matplotlib*). In the code below, *mark_circle* is used to indicate the type of chart desired (scatter plot with circles) and the *encode* function specify the chart encoding, in this case, what columns are mapped to the *x* and *y* positions, *color* of the circle, and *tooltip* on hover.

For more complex examples, see the Altair documentation. There are many chart possibilities, and

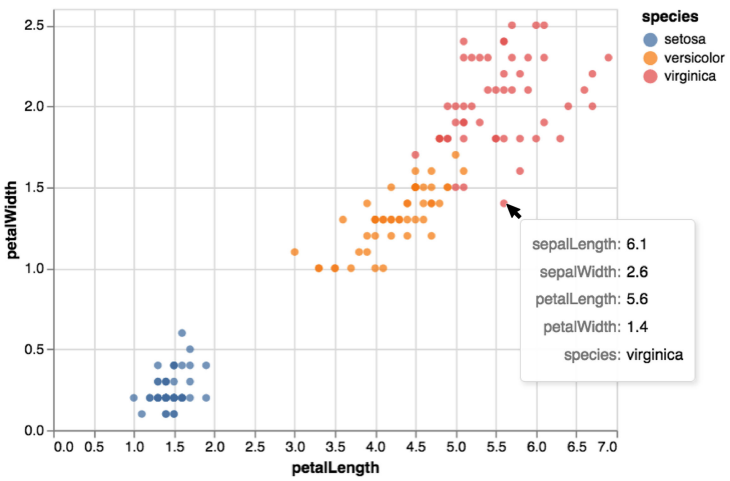

FIGURE 2. Interactive Altair scatter plot of the Iris dataset. The chart displays a tooltip with flower information on mouse hover. The library also enables pan and zoom.

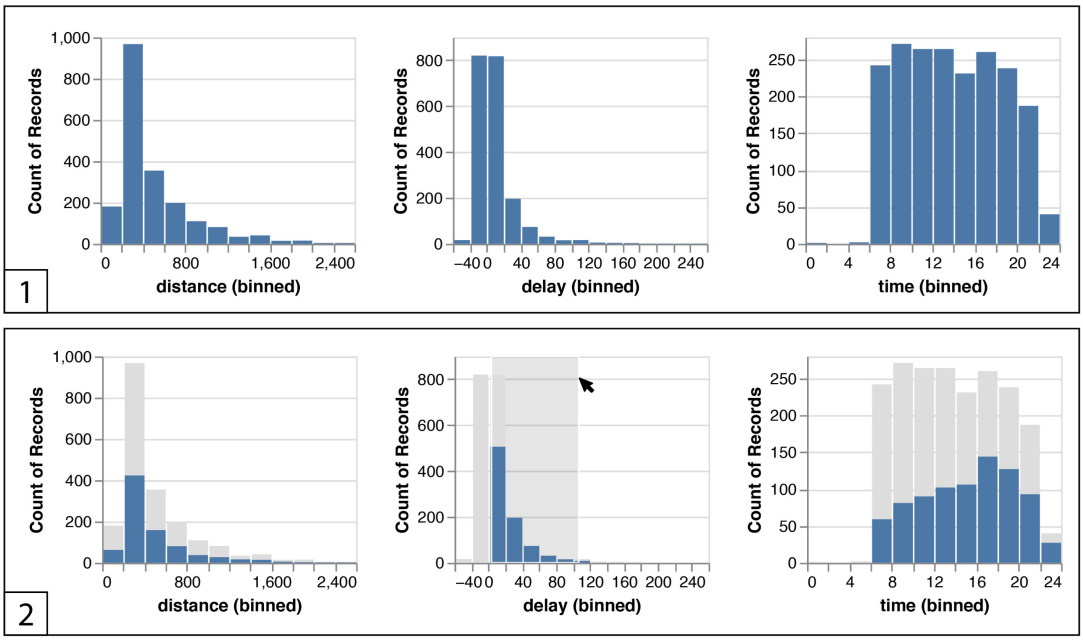

FIGURE 3. Altair Dashboard showing a flight dataset. (1) Histograms for flight distance, delay, and time. (2) User selected a range of delay values and the system automatically updates the other views.

graphics can be combined to create interactive dashboards with multiple views. For example, Figure 3(1) shows an Altair dashboard that visualizes a flight dataset [example taken from the online documentation (https://altair-viz.github.io/gallery/interactive_layered_crossfilter.html)]. (2) The user can select flights based on delay (in hours) and see how delay correlates with the other variables (distance and time).

One disadvantage of *Altair* is that we cannot have access to data generated by the user in Python. For example, we would not be able to receive data points selected in Altair in the next Jupyter cell. Such capability exists in *matplotlib* and in custom JavaScript visualizations, because we can set up callbacks between JavaScript and Python.

HTML Embedding

Displaying custom visualizations in a Jupyter Notebook can be done in a few lines of code using the package *Ipython.display*, which embeds HTML code in notebook cells. The HTML may contain both CSS and JavaScript, which affords flexible, interactive, and customizable visualizations to be created.

In order to embed the visualization in a cell, one needs to create a string variable containing all the HTML, JavaScript, data, and CSS code needed for the visualization. Since writing everything in a Jupyter cell can be too cumbersome, one can write the visualization in a code editor and then load the document in Python. JavaScript Bundlers, such as Webpack, can convert multiple HTML, JavaScript, and CSS files into a single file, facilitating this process.

In the following, we show an example of HTML embedding in a Jupyter cell. The code adds a single button to the page, which when clicked displays an alert box with the message "Hello World."

```
[ ]: from IPython.display import display, HTML
     html_string = """<button onclick="alert('Hello World')">Hello World</button>"""
     display(HTML(html_string))
```

Formatting methods can be used to create the HTML string. For example, a base string may contain the container *div* where the visualization is going to be inserted, a *script* tag where the bundled code is going to be added, and a function call to plot the visualization with the provided data in JSON format. The *string.format* function can be used to add the remaining information to the string, filling in the placeholders.

The following code snippet shows how to embed a JavaScript library and CSV data in the

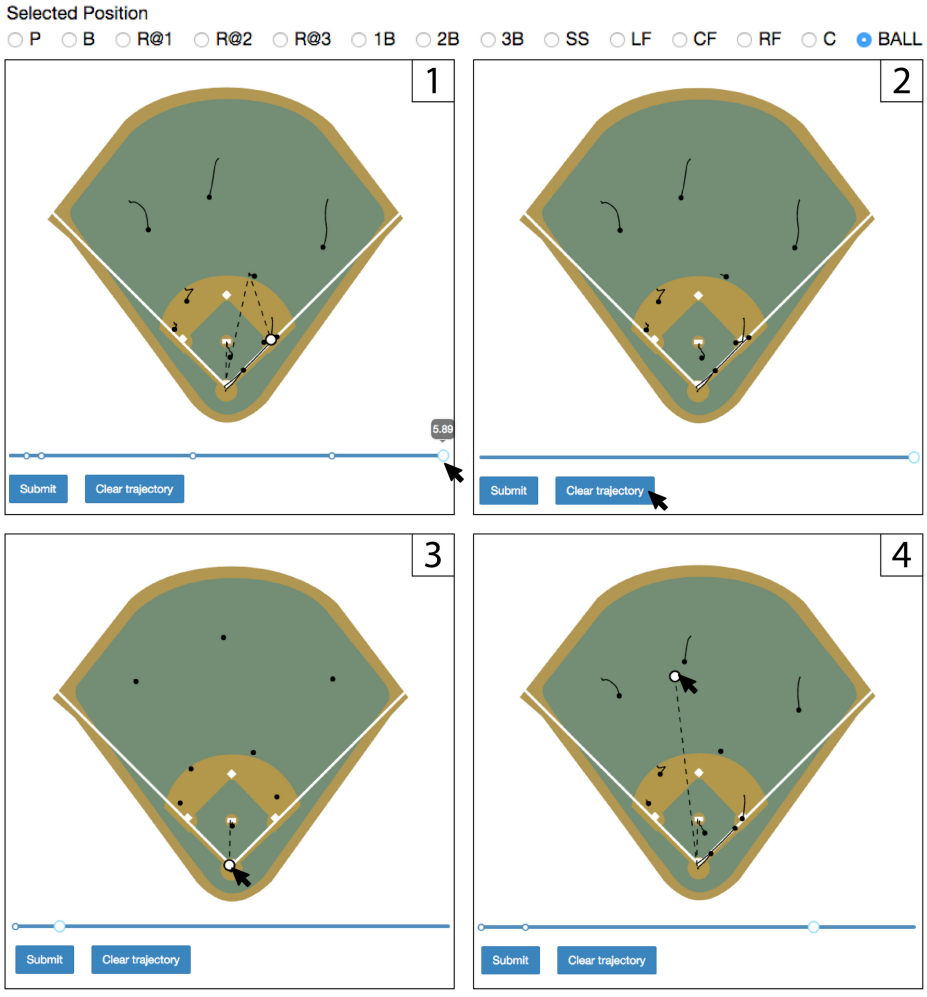

FIGURE 4. Custom JavaScript visualization of baseball plays. The user can (1) animate the play using the slider. Select a position to edit (in the picture, the BALL is selected) and (2) clear the trajectory. Annotate the positions of the ball when it is thrown (3) and hit to the center field (4).

HTML string. This example visualization shows an interactive chart that displays baseball game trajectories (see Figure 4). The user can control the progress of the play using a slider. Furthermore, the user can select a player or the ball to edit its trajectory (either clicking on the field, or the button "Clear trajectory"). This visualization is an adaptation of the Baseball annotation system HistoryTracker.[6]

```
[ ]: from IPython.display import display, HTML
import pandas as pd

with open("./BaseballVisualizer/build/baseballvisualizer.js", "r") as f:
    bundled_code = f.read()

play = pd.read_csv("./BaseballVisualizer/play_annotated.csv")
data = {'tracking': play.to_json(orient="records")}

html = """
<html>
<body>
<div id="container"/>
<script type="application/javascript">
    {bundled_code}
    baseballvisualizer.renderBaseballAnnotator("#container", {data});
</script>
</body>
</html>
""".format(bundled_code=bundled_code, data=data)

display(HTML(html))
```

Callbacks can be set up in both JavaScript and Python using the *comm* API (https://jupyter-notebook.readthedocs.io/en/stable/comms.html) to send data from one to the other. For example, if a sports analyst is interested in modifying a Baseball trajectory and running some further analysis in Python, he might set up this bidirectional communication.

A minor change needs to be made to both the JavaScript and to the Python code. In JavaScript, a new *comm* object needs to be created with an identifier (in this example, *submit_trajectory*). Then, the *comm* object is used to *send* a message to Python, containing the edited trajectory data. Finally, when Python acknowledges the message, we display an alert.

```
function submitTrajectoryToServer(trajectory){
    let comm = window.Jupyter.notebook.kernel.comm_manager.new_comm('submit_trajectory')
    // Send trajectory to Python
    comm.send({'trajectory': trajectory})

    // Receive message from Python
    comm.on_msg(function(msg) {
        alert("Trajectory received by Jupyter Notebook.")
    });
}
```

The Python code needs to expect a message from JavaScript. In order to set this up, we use the *register_target* function, passing to it the communication identifier and the Python callback function. In the following code snippet, this callback will store the trajectory in the variable *received_trajectory*.

```
[ ]: received_trajectory = []
     def receive_trajectory(comm, open_msg):
         # comm is the kernel Comm instance
         # Register handler for future messages
         @comm.on_msg
         def _recv(msg):
             global received_trajectory
             # Use msg['content']['data'] for the data in the message
             received_trajectory = msg['content']['data']['trajectory']
             print(received_trajectory)
             comm.send({'received': True})

     get_ipython().kernel.comm_manager.register_target('submit_trajectory',␣
      ↪receive_trajectory)
```

Finally, after the user clicks the "Submit" button, the trajectory can be retrieved in the Jupyter Notebook, analyzed and saved to disk.

```
[ ]: received_trajectory_df = pd.DataFrame(received_trajectory)
     received_trajectory_df.to_csv("edited_trajectory.csv", index=False)
```

TO LOOK FURTHER

There are many domain-specific visualization libraries for Jupyter Notebook, which use the techniques described in this article. Figure 5 shows examples in three different domains, which illustrate how diverse and flexible these visualizations can be. The examples belong to the fields of scientific visualization,[7] sports analytics,[8] and machine learning.[9,10] 1) *ipygany*[7] enables the visualization of 3-D meshes in Jupyter Notebooks. Users can zoom, rotate, and apply effects to 3-D meshes interactively using this library. 2) *StatCast Dashboard*[8] supports the interactive query, filter, and visualization of spatiotemporal baseball trajectories and statistics. The library communicates with a baseball play database in order to execute complex queries involving player, teams, game dates, and events. 3) *InterpretML*[9] is a Python package that contains a collection of algorithms for explaining and visualizing machine learning (ML) models, including LIME, SHAP, and partial dependence plots. Finally, 4) *PipelineProfiler*[10] contains visualizations that enable the exploration and comparison of ML pipelines produced by Automatic ML systems.

One of the advantages of Jupyter Notebooks is that they support reproducibility.[1] However, when interactive visualizations are used in computational notebooks, additional mechanisms are needed to afford reproducible results. We refer the interested reader to the work by Fekete and Freire[11] for a survey of reproducibility challenges faced by the visualization community and how those challenges can be addressed.

In this article, we have presented three ways to create interactive visualizations in Jupyter Notebooks: *matplotlib* charts, *Altair* specifications, and custom HTML visualizations. We hope that this document will help developers to create their own interactive charts.

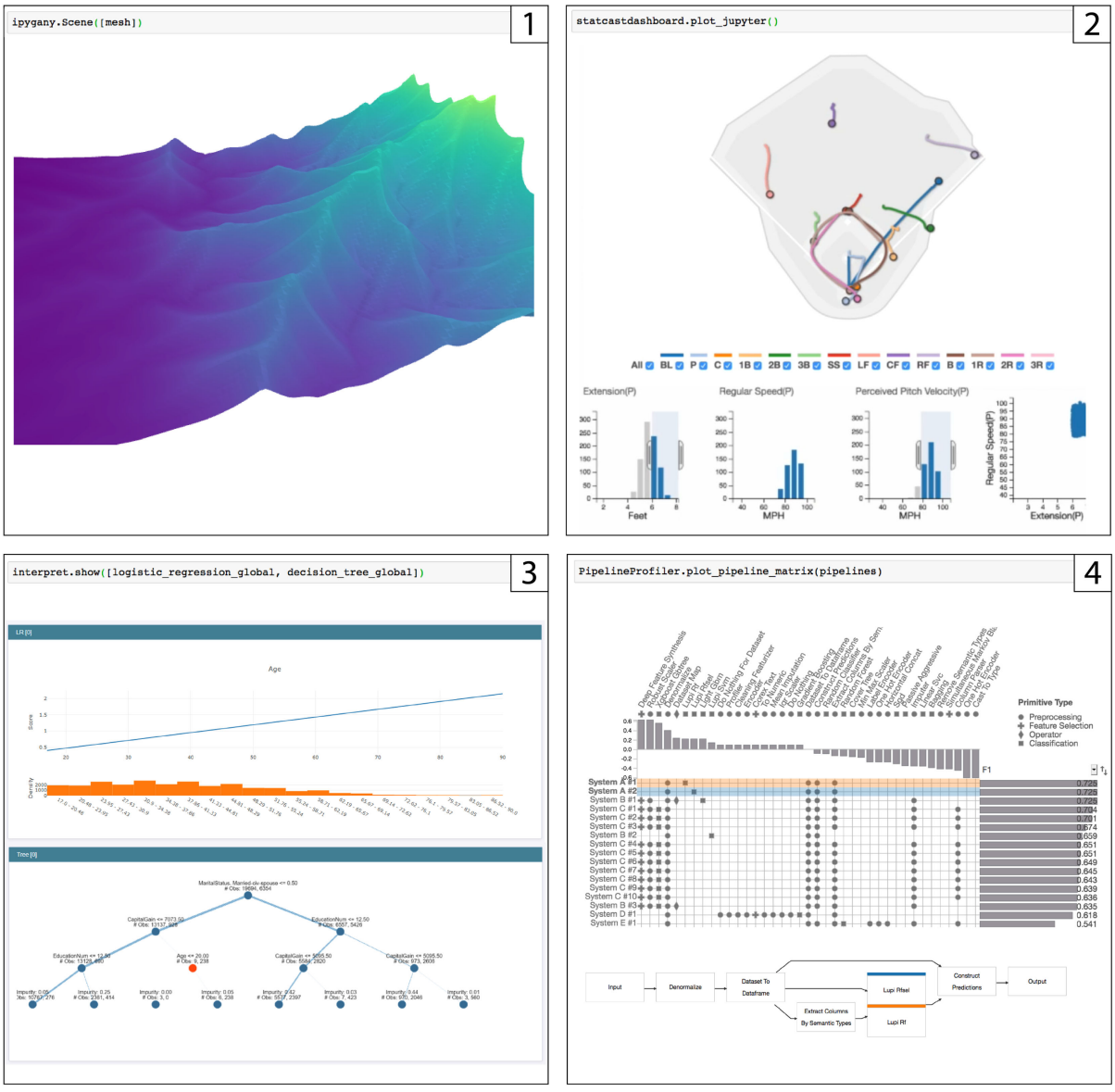

FIGURE 5. Domain-specific visualization libraries for Jupyter Notebook. 1) *ipygany:* visualization of 3-D meshes. 2) *StatCast Dashboard:* Visualization of baseball trajectories and game statistics. 3) *InterpretML:* Visualization of machine learning model explanations. 4) *PipelineProfiler:* Visualization of machine learning pipelines produced by AutoML systems.

ACKNOWLEDGMENTS

The authors would like to thank João Comba for his thoughtful comments that helped improve their article. This work was supported in part by the DARPA D3M program, in part by NASA, in part by Adobe, and in part by NSF awards CNS-1229185, CCF-1533564, CNS-1544753, CNS-1730396, and CNS-1828576. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of NSF and DARPA.

REFERENCES

1. J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter Notebooks," in *Proc. ACM 16th Int. Conf. Mining Softw. Repositories*, 2019, pp. 507–517, doi: 10.1109/MSR.2019.00077.
2. J. D. Hunter, "Matplotlib: A 2D graphics environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/MCSE.2007.55.

3. J. M. Perkel, "Data visualization tools drive interactivity and reproducibility in online publishing," *Nature*, vol. 554, no. 7690, pp. 133–134, 2018, doi: 10.1038/d41586-018-01322-9.
4. J. VanderPlas *et al.*, "Altair: Interactive statistical visualizations for Python," *J. Open Source Softw.*, vol. 3, no. 32, 2018, Art. no. 1057, doi: 10.21105/joss.01057.
5. A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-Lite: A grammar of interactive graphics," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 341–350, Jan. 2017, doi: 10.1109/TVCG.2016.2599030.
6. J. P. Ono, A. Gjoka, J. Salamon, C. Dietrich, and C. T. Silva, "HistoryTracker: Minimizing human interactions in baseball game annotation," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2019, pp. 1–12, doi: 10.1145/3290605.3300293.
7. M. Breddels, ipygany: Scientific Visualization in the Jupyter Notebook, 2020. [Online]. Available: https://ipygany.readthedocs.io/en/latest/
8. M. Lage, J. P. Ono, D. Cervone, J. Chiang, C. Dietrich, and C. T. Silva, "StatCast dashboard: Exploration of spatiotemporal baseball data," *IEEE Comput. Graph. Appl.*, vol. 36, no. 5, pp. 28–37, Sep./Oct. 2016, doi: 10.1109/MCG.2016.101.
9. H. Nori, S. Jenkins, P. Koch, and R. Caruana, "InterpretML: A unified framework for machine learning interpretability," 2019, *arXiv:1909.09223*.
10. J. P. Ono, S. Castelo, R. Lopez, E. Bertini, J. Freire, and C. Silva, "PipelineProfiler: A visual analytics tool for the exploration of AutoML pipelines," *IEEE Trans. Vis. Comput. Graph.*, vol. 27, no. 2, pp. 390–400, Feb. 2021, doi: 10.1109/TVCG.2020.3030361.
11. J.-D. Fekete and J. Freire, "Exploring reproducibility in visualization," *IEEE Comput. Graph. Appl.*, vol. 40, no. 5, pp. 108–119, Sep./Oct. 2020, doi: 10.1109/MCG.2020.3006412.

JORGE PIAZENTIN ONO is currently working toward the Ph.D. degree with New York University, New York, NY, USA. His research interests include visualization, graphics, human-computer interaction, sports analytics, and interactive machine learning. He received the M.S. degree in computer science from the University of Sao Paulo, Sao Carlos, SP, Brazil. Contact him at jorgehpo@nyu.edu.

JULIANA FREIRE is currently a professor of computer science and data science with New York University, New York, NY, USA. Her research interests span topics in large-scale data analysis, curation and integration, visualization, machine learning, provenance management, and web information discovery. She received the Ph.D. degree in computer science from the State University of New York at Stony Brook, Stony Brook, NY, USA. Contact her at juliana.freire@nyu.edu.

CLAUDIO T. SILVA is currently a professor of computer science and data science with New York University, New York, NY, USA. His research interests include visualization, graphics, geometric computing, sports analytics, and urban computing. He received the Ph.D. degree in computer science from the State University of New York at Stony Brook, Stony Brook, NY, USA. Contact him at csilva@nyu.edu.