The Journal of Open Source Software

seaborn: statistical data visualization

Michael L. Waskom[1]

1 Center for Neural Science, New York University

DOI: 10.21105/joss.03021

Software

- Review
- Repository
- Archive

Editor: Lorena Pantano

Reviewers:

- @dangeles
- @Sara-ShiHo

Submitted: 29 January 2021
Published: 06 April 2021

Summary

`seaborn` is a library for making statistical graphics in Python. It provides a high-level interface to `matplotlib` and integrates closely with `pandas` data structures. Functions in the `seaborn` library expose a declarative, dataset-oriented API that makes it easy to translate questions about data into graphics that can answer them. When given a dataset and a specification of the plot to make, `seaborn` automatically maps the data values to visual attributes such as color, size, or style, internally computes statistical transformations, and decorates the plot with informative axis labels and a legend. Many `seaborn` functions can generate figures with multiple panels that elicit comparisons between conditional subsets of data or across different pairings of variables in a dataset. `seaborn` is designed to be useful throughout the lifecycle of a scientific project. By producing complete graphics from a single function call with minimal arguments, `seaborn` facilitates rapid prototyping and exploratory data analysis. And by offering extensive options for customization, along with exposing the underlying `matplotlib` objects, it can be used to create polished, publication-quality figures.

Statement of need

Data visualization is an indispensable part of the scientific process. Effective visualizations will allow a scientist both to understand their own data and to communicate their insights to others (Tukey, 1977). These goals can be furthered by tools for specifying a graph that provide a good balance between efficiency and flexibility. Within the scientific Python ecosystem, the `matplotlib` (Hunter, 2007) project is very well established, having been under continuous development for nearly two decades. It is highly flexible, offering fine-grained control over the placement and visual appearance of objects in a plot. It can be used interactively through GUI applications, and it can output graphics to a wide range of static formats. Yet its relatively low-level API can make some common tasks cumbersome to perform. For example, creating a scatter plot where the marker size represents a numeric variable and the marker shape represents a categorical variable requires one to transform the size values to graphical units and to loop over the categorical levels, separately invoking a plotting function for each marker type.

The `seaborn` library offers an interface to `matplotlib` that permits rapid data exploration and prototyping of visualizations while retaining much of the flexibility and stability that are necessary to produce publication-quality graphics. It is domain-general and can be used to visualize a wide range of datasets that are well-represented within a tabular format.

Example

The following example demonstrates the creation of a figure with `seaborn`. The example makes use of one of the built-in datasets that are provided for documentation and generation of

reproducible bug reports. It illustrates several of the features described in the Overview section, including the declarative API, semantic mappings, faceting across subplots, aggregation with error bars, and visual theme control.

```
import seaborn as sns
sns.set_theme(context="paper")
fmri = sns.load_dataset("fmri")
g = sns.relplot(
    data=fmri, kind="line",
    x="timepoint", y="signal",
    hue="event", style="event", col="region",
    height=3.5, aspect=.8,
)
g.savefig("paper_demo.pdf")
```

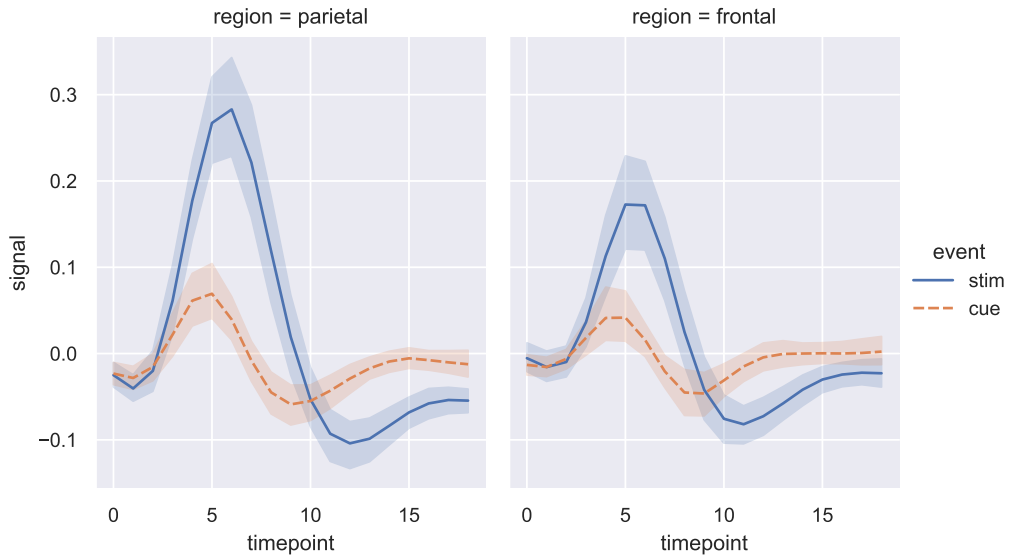

Figure 1: An example `seaborn` figure demonstrating some of its key features. The image was generated using seaborn v0.11.1.

Overview

Users interface with `seaborn` through a collection of plotting functions that share a common API for plot specification and offer many more specific options for customization. These functions range from basic plot types such as scatter and line plots to functions that apply various transformations and abstractions, such as histogram binning, kernel density estimation, and regression model fitting. Functions in `seaborn` are classified as either "axes-level" or "figure-level." Axes-level functions behave like most plotting functions in the `matplotlib.pyplot` namespace. By default, they hook into the state machine that tracks a "current" figure and add a layer to it, but they can also accept a matplotlib axes object to control where the plot is drawn, similar to using the `matplotlib` "object-oriented" interface. Figure-level functions create their own figure when invoked, allowing them to "facet" the dataset by creating multiple conditional subplots, along with adding conveniences such as putting the legend outside the space of the plot by default. Each figure-level function corresponds to several axes-level functions that serve similar purposes, with a single parameter selecting

the kind of plot to make. For example, the `displot` function can produce several different representations of a distribution, including a histogram, kernel density estimate, or empirical cumulative distribution function. The figure-level functions make use of a `seaborn` class that controls the layout of the figure, mediating between the axes-level functions and matplotlib. These classes are part of the public API and can be used directly for advanced applications.

One of the key features in `seaborn` is that variables in a dataset can be automatically "mapped" to visual attributes of the graph. These transformations are referred to as "semantic" mappings because they endow the attributes with meaning *vis a vis* the dataset. By freeing the user from manually specifying the transformations – which often requires looping and multiple function invocations when using `matplotlib` directly – `seaborn` allows rapid exploration of multidimensional relationships. To further aid efficiency, the default parameters of the mappings are opinionated. For example, when mapping the color of the elements in a plot, `seaborn` infers whether to use a qualitative or quantitative mapping based on whether the input data are categorical or numeric. This behavior can be further configured or even overridden by setting additional parameters of each plotting function.

Several `seaborn` functions also apply statistical transformations to the input data before plotting, ranging from estimating the mean or median to fitting a general linear model. When data are transformed in this way, `seaborn` automatically computes and shows error bars to provide a visual cue about the uncertainty of the estimate. Unlike many graphical libraries, `seaborn` shows 95% confidence interval error bars by default, rather than standard errors. The confidence intervals are computed with a bootstrap algorithm, allowing them to generalize over many different statistics, and the default level allows the user to perform "inference by eye" (Cumming & Finch, 2005). Historically, error bar specification has been relatively limited, but a forthcoming release (v0.12) will introduce a new configuration system that makes it possible to show nonparametric percentile intervals and scaled analytic estimates of standard error or standard deviation statistics.

`seaborn` aims to be flexible about the format of its input data. The most convenient usage pattern provides a `pandas` (McKinney, 2010) dataframe with variables encoded in a long-form or "tidy" (Wickham, 2014) format. With this format, columns in the dataframe can be explicitly assigned to roles in the plot, such as specifying the `x` and `y` positions of a scatterplot along with size and shape semantics. Long-form data supports efficient exploration and prototyping because variables can be assigned different roles in the plot without modifying anything about the original dataset. But most `seaborn` functions can also consume and visualize "wide-form" data, typically producing similar output to how the analogous `matplotlib` function would interpret a 2D array (e.g., producing a boxplot where each box represents a column in the dataframe) while making use of the index and column names to label the graph. Using the label information in a `pandas` object can help make plots that are interpretable without further tweaking – reducing the chance of interpretive errors – but `seaborn` also accepts data from a variety of more basic formats, including `numpy` (Harris et al., 2020) arrays and simple Python collection types.

`seaborn` also offers multiple built-in themes that users can select to modify the visual appearance of their graphs. The themes make use of the `matplotlib` `rcParams` system, meaning that they will take effect for any figure created using `matplotlib`, not just those made by `seaborn`. The themes are defined by two disjoint sets of parameters that separately control the style of the figure and the scaling of its elements (such as line widths and font sizes). This separation makes it easy to generate multiple versions of a figure that are scaled for different contexts, such as written reports and slide presentations. The theming system can also be used to set a default color palette. As color is particularly important in data visualization and no single set of defaults is universally appropriate, every plotting function makes it easy to choose an alternate categorical palette or continuous gradient mapping that is well-suited for the particular dataset and plot type. The seaborn documentation contains a tutorial on the use of color in data visualization to help users make this important decision.

`seaborn` does not aim to completely encapsulate or replace `matplotlib`. Many useful

graphs can be created through the `seaborn` interface, but more advanced applications – such as defining composite figures with multiple arbitrary plot types – will require importing and using `matplotlib` as well. Even when calling only `seaborn` functions, deeper customization of the plot appearance is achieved by specifying parameters that are passed-through to the underlying `matplotlib` functions, and tweaks to the default axis limits, ticks, and labels are made by calling methods on the `matplotlib` object that axes-level `seaborn` functions return. This approach is distinct from other statistical graphing systems, such as `ggplot2` (Wickham, 2016). While `seaborn` offers some similar features and, in some cases, uses similar terminology to `ggplot2`, it does not implement the formal Grammar of Graphics and cannot be used to produce arbitrary visualizations. Rather, its aim is to facilitate rapid exploration and prototyping through named functions and opinionated defaults while allowing the user to leverage the considerable flexibility of `matplotlib` to create more domain-specific graphics and to polish figures for publication. An example of a successful use of this approach to produce reproducible figures can be found at https://github.com/WagnerLabPapers/Waskom_PNAS_2017 (Waskom & Wagner, 2017).

Acknowledgements

M.L.W. has been supported by the National Science Foundation IGERT program (0801700) and by the Simons Foundation as a Junior Fellow in the Simons Society of Fellows (527794). Many others have helped improve `seaborn` by asking questions, reporting bugs, and contributing code; thank you to this community.

References

Cumming, G., & Finch, S. (2005). Inference by eye: confidence intervals and how to read pictures of data. *The American Psychologist*, *60*(2), 170–180. https://doi.org/10.1037/0003-066X.60.2.170

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., R'ıo, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

McKinney, W. (2010). Data structures for statistical computing in python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 51–56). https://doi.org/10.25080/Majora-92bf1922-00a

Tukey, J. W. (1977). *Exploratory data analysis*. Addison-Wesley. ISBN: 978-0201076165

Waskom, M. L., & Wagner, A. D. (2017). Distributed representation of context by intrinsic subnetworks in prefrontal cortex. *Proceedings of the National Academy of Sciences*, 2030–2035. https://doi.org/10.1073/pnas.1615269114

Wickham, H. (2014). Tidy data. *Journal of Statistical Software, Articles*, *59*(10), 1–23. https://doi.org/10.18637/jss.v059.i10

Wickham, H. (2016). *ggplot2: Elegant graphics for data analysis*. Springer-Verlag. ISBN: 978-3-319-24277-4