Swizz: One-Liner Figures, LaTeX Tables, and Flexible Layouts for Scientific Papers

Lars Quaedvlieg [*1] **Andrea Miele** [*1] **Caglar Gulcehre** [1]

Abstract

Producing publication-quality visualizations and tables for machine learning papers is often tedious, time-consuming, and prone to inconsistencies. We introduce Swizz, a lightweight Python library designed specifically for researchers to effortlessly generate elegant figures, LaTeX-ready tables, and customizable figure layouts with minimal code. Swizz enables one-line creation of consistent, conference-ready visualizations, including advanced plots and multilevel tables, and provides intuitive, composable layouts to simplify complex figure arrangements. Its automated styling and built-in visual gallery facilitate rapid experimentation, allowing researchers to focus more on research and less on formatting. Swizz is publicly available, easy to integrate into existing workflows, and is themed for major machine learning publication venues. Swizz is open source (MIT) and is available on GitHub and PyPI.

1. Introduction

With 45,758 papers submitted to the machine learning (ML) conferences NeurIPS, ICML, ICLR, and CVPR in 2024 alone, thousands of researchers collectively spend countless hours creating plots, formatting tables, and arranging complex figure layouts. This repeated manual work is tedious and distracts researchers from their main goal of advancing scientific knowledge. A unified, community-driven repository would eliminate redundant work and significantly reduce overhead.

We introduce `Swizz`, a lightweight Python library explicitly designed to become such a community-driven platform. `Swizz` provides intuitive one-liner APIs for creating elegant, publication-quality plots, tables, and flexible figure layouts. Most importantly, `Swizz` is designed for easy contributions to plot and table templates by the community, facilitating the rapid growth of a large collection of ready-to-use visualizations. Researchers can simply browse the visual gallery, pick the desired plot or table style, and instantly generate consistent, publication-ready outputs.

By minimizing the repetitive overhead of figure and table creation, `Swizz` empowers researchers to focus their valuable time on experimentation and scientific discovery rather than formatting. The library is fully open source (MIT), easily accessible via GitHub and PyPI, and we actively encourage contributions from the broader community[1]. Through collective effort, `Swizz` can continue to evolve as the go-to resource for visual presentation in ML publications.

1.1. Key Features

- **One-line plotting API**: Produces styled visualizations from raw data in one line of code.
- **Automatic LaTeX tables**: Convert raw data into ready-to-paste LaTeX tabular code.
- **Subfigure layouts**: Build complex figure grids with simple layout utilities.
- **Consistent styling**: Built-in color palettes, hatch patterns, and typography ensure consistent outputs between plots.

2. Related Works

`Swizz` builds directly on the well-known Matplotlib (Hunter, 2007) and Seaborn (Waskom, 2021) libraries, using their plotting APIs and styles while hiding the boilerplate code. Matplotlib remains the standard for fine-grained control in Python, and Seaborn adds high-level interfaces for statistical graphics; `Swizz` complements both by providing one-line wrappers and a unified styling layer for academic publications.

Several other tools aim to simplify visualization and layout in Python. Libraries like Plotly (Plotly Technologies, 2015)

[*]Equal contribution [1]EPFL, Switzerland. Correspondence to: Lars Quaedvlieg <larsquaedvlieg@outlook.com>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25). Copyright 2025 by the author(s).

[1]Join the development at https://github.com/lars-quaedvlieg/swizz

and Bokeh (Bokeh Development Team, 2018) offer interactive, web-friendly plots, but require a different API and often more setup for publication-quality figures. High-level interfaces such as Altair (VanderPlas et al., 2018) provide concise Grammar of Graphics abstractions, but they do not integrate directly with LaTeX table generation or multipanel figure composition. For tabular output, Pandas' (Wes McKinney, 2010) 'DataFrame.to_latex' and third-party packages like Tabulate (Tabulate Team, 2022) automate basic table conversion, yet lack built-in support for multilevel headers, custom formatting, and automatic highlighting of best entries – a gap that `Swizz` fills.

Matplotlib's built-in grid tools (e.g. 'GridSpec' and 'subplot_mosaic') let you define arbitrary subplot arrangements, but they remain fairly low-level: You must handcraft each grid spec or mosaic pattern, manually hide or show axis labels, and tweak legend placement and padding by hand. `Swizz` builds on these foundations but exposes a **Flutter-inspired block layout** model–'PlotBlock', 'Row/Col', 'LegendBlock', 'Label', etc.–so you can simply say "2×3 grid with shared legend and bottom-row x-labels" (or any other combination) and let the library handle axis visibility, legend collection, internal padding, and annotation blocks automatically. Some higher-level frameworks (e.g. HoloViews (Rudiger et al., 2020) or Plotly's Dash) enable dashboard-style layouts, but they target interactive environments rather than static, publication-ready panels.

We provide a table of differences between these libraries in Appendix D. By combining one-liner plot wrappers, automated LaTeX table formatting, and composable layout blocks, `Swizz` forms a library for researchers who need both the programmatic simplicity and the polish required to write high-quality papers.

3. Architecture and Modules

`Swizz` is organized into three core modules:

`swizz.plot` A collection of one-line plotting functions for standard chart types (lines with error bands, histograms, bar charts, scatter/UMAP/t-SNE, etc.). Each function applies consistent, publication-ready styling (colors, fonts, grids) automatically.

`swizz.table` Utilities to convert raw data (e.g. `pandas.DataFrame`) into LaTeX tabular code. Built-in support for multi-level headers, mean ± std (or stderr) formatting, percentage display, and flexible row/column grouping.

`swizz.layout` A Flutter-inspired block layout API for composing multipanel figures. Core building blocks include `PlotBlock`, `Row`, `Col`, `LegendBlock`, and `Label`. Blocks can be nested arbitrarily, and predefined layouts (e.g. `grid_stack` for an $n \times m$ grid with shared legends and selective axis labels) are provided for common use cases.

Each module focuses on a single concern, but they interoperate seamlessly: plots produce Matplotlib axes that feed into layout blocks, and tables emit LaTeX code ready for direct insertion into the paper.

3.1. Styling and Themes

`Swizz` applies visual themes by combining Matplotlib's `rcParams`, style sheets, and Seaborn palettes. When calling `set_style`, it resets defaults, applies the selected theme's style (e.g., `seaborn-v0_8-whitegrid`), sets the palette, and updates `rcParams` for fonts, grids, and figure settings—ensuring consistent plots with minimal user effort. It also means that users are able to load their own style sheets if they prefer to use custom styles instead.

4. Usage Example

In the following, we demonstrate an example of a call to each module in turn, showcasing how easy it is to generate paper-ready tables and plots from your data. Furthermore, in Appendix B, we showcase the difference in complexity to produce plots with our package versus Matplotlib.

4.1. Tables

In Appendix A, Snippet 1 demonstrates how a single call to `swizz.table` automatically produces all of the formatting, grouping, and highlighting logic. The resulting LaTeX code can be pasted directly into your paper. In the table gallery of the documentation, it also tells you which packages and commands need to be used in order to compile the table. Table 1 shows the output of the code in Snippet 1.

Table 1. Example table: expert-normalized returns across domains and methods.

Domain	Task	Component	MLP	Modality
Ant	Dynamics change (↑) [4]	$\mathbf{1.01 \pm 0.07}$	1.00 ± 0.02	0.97 ± 0.04
	IL (↑) [1]	$\mathbf{0.98 \pm 0.04}$	0.96 ± 0.05	0.98 ± 0.06
	Off-RL (↑) [1]	0.98 ± 0.04	$\mathbf{1.05 \pm 0.04}$	1.02 ± 0.03
	Sensor failure (↑) [11]	0.97 ± 0.04	1.01 ± 0.04	$\mathbf{1.02 \pm 0.05}$
HalfCheetah	Dynamics change (↑) [4]	$\mathbf{1.00 \pm 0.02}$	0.97 ± 0.05	0.98 ± 0.03
	IL (↑) [1]	1.03 ± 0.03	$\mathbf{1.07 \pm 0.04}$	1.00 ± 0.04
	Off-RL (↑) [1]	$\mathbf{1.01 \pm 0.06}$	1.01 ± 0.04	1.00 ± 0.09
	Sensor failure (↑) [11]	0.97 ± 0.07	0.97 ± 0.05	$\mathbf{1.02 \pm 0.04}$
Hopper	Dynamics change (↑) [4]	1.03 ± 0.02	$\mathbf{1.04 \pm 0.07}$	1.04 ± 0.06
	IL (↑) [1]	$\mathbf{1.01 \pm 0.05}$	0.97 ± 0.03	0.98 ± 0.04
	Off-RL (↑) [1]	$\mathbf{1.03 \pm 0.06}$	0.98 ± 0.02	1.00 ± 0.07
	Sensor failure (↑) [11]	0.99 ± 0.08	1.02 ± 0.03	$\mathbf{1.03 \pm 0.04}$
Walker2d	Dynamics change (↑) [4]	0.99 ± 0.03	1.00 ± 0.06	$\mathbf{1.00 \pm 0.04}$
	IL (↑) [1]	0.98 ± 0.03	1.00 ± 0.05	$\mathbf{1.02 \pm 0.04}$
	Off-RL (↑) [1]	0.99 ± 0.06	1.00 ± 0.04	$\mathbf{1.02 \pm 0.07}$
	Sensor failure (↑) [11]	$\mathbf{1.01 \pm 0.06}$	1.00 ± 0.05	0.98 ± 0.05

A complete gallery of all supported table styles can be found in the Swizz documentation: https://lars-quaedvlieg.github.io/swizz/tables/index.html.

4.2. Figures

Swizz also offers multiple built-in plot "themes" (e.g. latex, dark_latex, nature, etc.). You can preview all of them at https://lars-quaedvlieg.github.io/swizz/plot_themes/index.html.

In Appendix A, Snippet 2 also shows how just two calls to swizz.plot create publication-ready charts with error bands or percentile curves. In Figure 1 and Figure 2 we display the rendered outputs.

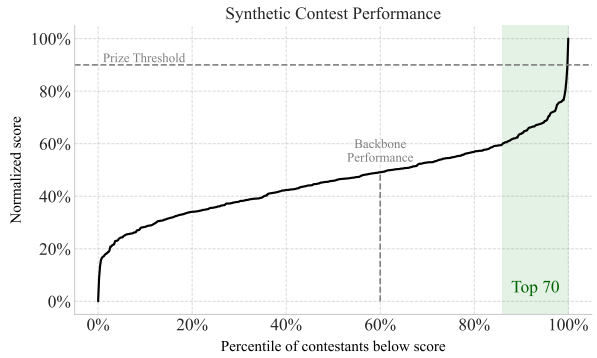


Figure 1. Percentile curve plot illustrating the distribution of values across percentiles.

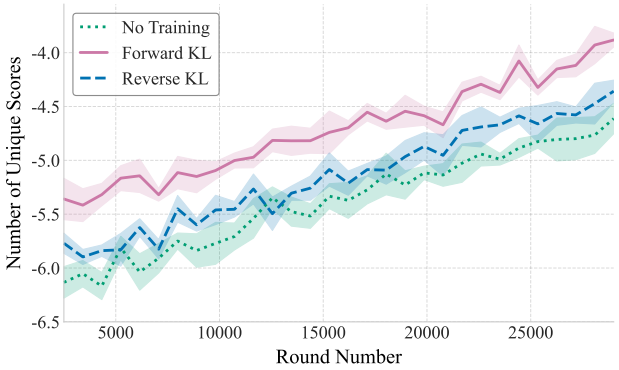


Figure 2. Multiple standard-deviation lines showing uncertainty bands around mean trajectories.

A full gallery of Swizz figure types is available at: https://lars-quaedvlieg.github.io/swizz/plots/index.html.

4.3. Figure Layouts

Swizz's layout module lets you assemble complex, publication-ready figures from simple building blocks. Core blocks include PlotBlock, Row, Col, LegendBlock, and Label.

Snippet 3 shows how to compose a nested layout with two rows of shared legends and labeled sub-panels. The rendered result appears in Figure 3.

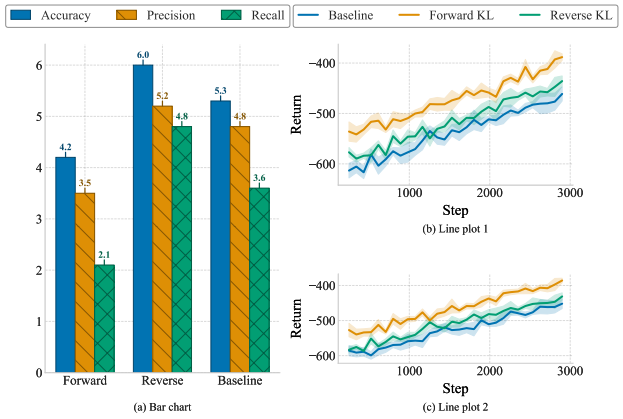


Figure 3. Nested layout with two legend rows and three labeled subplots, generated by the code in Listing 3.

Similarly, Snippet 4 demonstrates how to build a 2×3 grid of bar plots with selective axis labels and custom widths using the grid_stack preset, and Figure 4 shows the resulting arrangement.

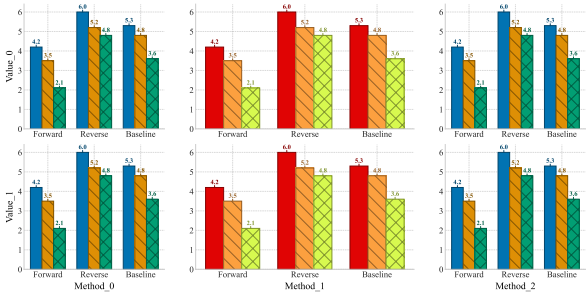


Figure 4. Grid layout generated by grid_stack as shown in Listing 4.

These examples illustrate how Swizz handles legends, labels, spacing, and margins automatically. For more layouts and detailed examples, see the full gallery at https://lars-quaedvlieg.github.io/swizz/intro.html.

5. Documentation and Community

Comprehensive documentation, live examples, and API references are available at https://lars-quaedvlieg.github.io/swizz/. Contributions, issue reports, and pull requests are welcome through the GitHub repository, with the guidelines outlined in CONTRIBUTING.md.

5.1. Contribution Workflow

Contributing new plots or tables to Swizz is designed to be as simple and streamlined as possible. The workflow below ensures that your addition is automatically integrated into the documentation and usable with the standard API.

1. **Create Your Module:** Add a new Python file under the appropriate directory:
   - Plots: swizz/plots/yourfile.py
   - Tables: swizz/tables/yourfile.py
2. **Write the Plot or Table Function:** Define your plot or table generator. The function should be self-contained and return a standard Matplotlib figure or LaTeX-formatted string.
3. **Register the Function:** Use the @register_plot or @register_table decorator to register your function. This ensures your contribution is automatically recognized by the library and added to the documentation gallery.

   Example (for a plot):

```python
@register_plot(
    name="simple_bar_plot",
    description="A simple bar plot with three bars"
                " and value annotations.",
    example_image="simple_bar_plot.png",
    example_code="simple_bar_plot.py"
)
def simple_bar_plot():
    ...
    return fig, ax
```

4. **Add an Example Snippet:** Create a Python example script that calls your function and saves the output. Place it under:
   - docs/_static/snippets/plots/ or .../tables/
5. **Render the Output:** Run the script to generate the output (plot image or LaTeX table), and save it to:
   - docs/_static/images/plots/ or .../tables/
6. **Automatic Documentation:** You do not need to manually modify the docs. The Jupyter Book automatically detects all registered plots/tables and builds a live gallery.
7. **Submit a Pull Request:** Fork the repository, push your new branch (e.g., feature/new-bar-plot), and open a pull request. Be sure to describe your contribution clearly.

For more details and examples, refer to the CONTRIBUTING.md file in the repository.

5.2. Limitations

Although Swizz simplifies the creation of publication-ready figures and tables, there are a few limitations in this first release. First, customization beyond the provided themes and presets may require additional effort. Although users receive the underlying Matplotlib figure object, which enables post-hoc edits, deep modifications may necessitate dropping into lower-level code. Second, interactive visualizations (e.g., using Plotly for instance) are currently unsupported, as Swizz focuses on static, publication-quality outputs. Lastly, not all plot types are supported yet, but we are actively expanding the library's capabilities with new templates and welcome community contributions to accelerate this process, as explained in 5.1.

6. Conclusion

Swizz simplifies the process of creating publication-quality figures, LaTeX tables, and multipanel layouts with minimal code. By combining one-line plot wrappers, automated table formatting, and a composable block-based layout API, Swizz enables researchers to focus on insights rather than styling. We invite the community to contribute new plot types, table styles, and layout presets to continue evolving Swizz into the go-to toolkit for ML and data-science publications.

References

Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2018. URL https://bokeh.pydata.org/en/latest/.

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

Plotly Technologies. *Collaborative data science*. Montreal, QC, 2015. URL https://plot.ly.

Rudiger, P., Stevens, J.-L., Bednar, J. A., Nijholt, B., Andrew, B, C., Randelhoff, A., Mease, J., Tenner, V., maxalbert, Kaiser, M., ea42gh, Samuels, J., stonebig, LB, F., Tolmie, A., Stephan, D., Lowe, S., Bampton, J., kbowen, et al. holoviz/holoviews: Version 1.13.3 (v1.13.3), 2020. URL https://doi.org/10.5281/zenodo.3904606.

Tabulate Team. tabulate — pypi.org. https://pypi.org/project/tabulate/, 2022.

VanderPlas, J., Granger, B., Heer, J., Moritz, D., Wongsuphasawat, K., Satyanarayan, A., Lees, E., Timofeev, I., Welsh, B., and Sievert, S. Altair: Interactive statistical visualizations for python. *Journal of Open Source Software*, 3(32):1057, 2018. doi: 10.21105/joss.01057. URL https://doi.org/10.21105/joss.01057.

Waskom, M. L. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. doi: 10.21105/joss.03021. URL https://doi.org/10.21105/joss.03021.

Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

A. Snippets

A.1. Tables

```python
import numpy as np
import pandas as pd
from swizz import table

### Generating the data
np.random.seed(0)

domains = ["HalfCheetah", "Hopper", "Walker2d", "Ant"]
tasks = ["IL (↑) [1]", "Off-RL (↑) [1]", "Sensor failure (↑) [11]", "Dynamics change (↑) [4]"]
methods = ["MLP", "Modality", "Component"]

rows = []
for domain in domains:
    for task in tasks:
        for method in methods:
            values = np.round(np.random.normal(loc=1.0, scale=0.05, size=5), 3).tolist()
            rows.append({
                "Domain": domain,
                "Task": task,
                "Method": method,
                "score": values
            })

df = pd.DataFrame(rows)

### Making the LaTeX for the table
latex = table(
    "grouped_multirow_latex",
    df=df,
    row1="Domain",
    row2="Task",
    col="Method",
    value_column="score",
    highlight="max",
    stderr=False,
    caption="Example table: expert-normalized returns across domains and methods.",
    label="tab:tokenization_comparison"
)
print(latex)
```

Snippet 1. Generate a multi-row, multi-column LaTeX table in one call.

A.2. Figures

```python
import numpy as np
import matplotlib.pyplot as plt

from swizz import plot, set_style

# Choose a theme ("latex" is default)
set_style("latex")

# Generate some dummy data
rounds = np.linspace(250, 2900, 30)

def fake_curve(seed, offset=0):
    np.random.seed(seed)
    base = np.linspace(-550 + offset, -400 + offset, len(rounds))
    noise = np.random.normal(0, 8, size=len(rounds))
    stderr = np.random.uniform(5, 20, size=len(rounds))
    return base + noise, stderr

averaged_metrics = {
    "forward-method": {
        "round_num": rounds,
        "unique_scores": fake_curve(0)[0],
        "std_error": fake_curve(0)[1],
    },
    "reverse-method": {
        "round_num": rounds,
        "unique_scores": fake_curve(1, -40)[0],
        "std_error": fake_curve(1)[1],
```

```
    },
    "baseline": {
        "round_num": rounds,
        "unique_scores": fake_curve(2, -60)[0],
        "std_error": fake_curve(2)[1],
    },
}

# Generate fake scores
np.random.seed(42)
scores = np.random.normal(200, 400, size=500)

fig, ax = plot("percentile_curve_plot",
    scores=scores,
    normalize_scores=True,
    normalize_percentiles=True,
    horizontal_markers=[
        (0.9, "Prize Threshold"),
    ],
    vertical_markers=[
        (0.6, "Backbone\nPerformance"),
    ],
    highlight_top_n=70,
    highlight_label="Top 70",
    highlight_label_color="darkgreen",
    highlight_label_font_size=16,
    highlight_color="#c8e6c9",
    vertical_label_offset=0.03,
    xlabel="Percentile of contestants below score",
    ylabel="Normalized score",
    title="Synthetic Contest Performance",
    font_family="Times New Roman",
    font_axis=14,
    figsize=(8, 5),
)
plt.show()

fig, ax = plot(
    "multiple_std_lines",
    data_dict=averaged_metrics,
    label_map={
        "forward-method": "Forward KL",
        "reverse-method": "Reverse KL",
        "baseline": "No Training",
    },
    style_map={
        "forward-method": "solid",
        "reverse-method": "dashed",
        "baseline": "dotted",
    },
    color_map={
        "forward-method": "#CC79A7",
        "reverse-method": "#0072B2",
        "baseline": "#009E73",
    },
    xlabel="Round Number",
    ylabel="Number of Unique Scores",
    xlim=(250, 2900),
    ylim=(-650, -355),
    x_formatter=lambda x, _: f"{x * 10:.0f}",
    y_formatter=lambda y, _: f"{y / 100:.1f}",
    save="ablation"
)

plt.show()
```

Snippet 2. One-line calls to produce styled figures under the chosen theme.

A.3. Figure Layout

```
from swizz.layouts.blocks import LegendBlock, Label
from swizz.layouts import render_layout
nested_layout = Col([
    Row([
        LegendBlock(
            labels=["Accuracy", "Precision", "Recall"],
            ncol=3, fixed_width=0.35),
        LegendBlock(
```

```
            labels=["Forward KL", "Reverse KL"],
            ncol=2)
    ], fixed_height=0.08, spacing=0.15),
    Row([
        Col([
            plot3,
            Label("(a) Bar chart", align="center",
                  fixed_height=0.05),
        ]),
        Col([
            plot1,
            Label("(b) Line plot 1", align="center",
                  fixed_height=0.05),
            plot2,
            Label("(c) Line plot 2", align="center",
                  fixed_height=0.05)
        ], spacing=0.07)
    ], spacing=0.1),
], spacing=0.02)

fig = render_layout(nested_layout, figsize=(10, 8))
plt.show()
```

Snippet 3. Compose a nested layout with shared legends and labeled sub-panels.

```
from swizz import layout
from swizz.layouts import render_layout, PlotBlock
import matplotlib.pyplot as plt

# Define grid size
n_rows, n_cols = 2, 3

# Defines the way to place plots within the grid and their arguments
def plot_fn(row_idx, col_idx):
    xlabel = ylabel = None
    fixed_width = None
    if col_idx == 1:
        fixed_width = 0.35
    if row_idx == n_rows - 1:
        xlabel = f"Method_{col_idx}"
    if col_idx == 0:
        ylabel = f"Value_{row_idx}"

    return PlotBlock("general_bar_plot",
                     fixed_width=fixed_width,
                     kwargs={
                         "data_dict": data_dict,
                         "style_map": style_map,
                         "xlabel": xlabel,
                         "ylabel": ylabel,
                         "color_map": None if col_idx != 1 else other_color_map,
                         "legend_loc": None}
    )

grid_layout = layout("grid_stack",
                     n_rows=2, n_cols=3,
                     plot_fn=plot_fn)

fig = render_layout(grid_layout,
                    figsize=(16, 8),
                    margins=(0.05, 0.1, 0.05, 0.05))
plt.show()
```

Snippet 4. Build a 2×3 grid of bar plots with selective axis labels and custom widths.

B. Matplotlib code versus Swizz

In this section, we compare the simplicity of Swizz with the equivalent Matplotlib code for a grouped bar plot.

B.1. Swizz (One-liner API)

```
import numpy as np
from matplotlib import pyplot as plt
```

```python
from swizz import plot

data_dict = {
    "Forward": {"Accuracy": 4.2, "Precision": 3.5, "Recall": 2.1},
    "Reverse": {"Accuracy": 6.0, "Precision": 5.2, "Recall": 4.8},
    "Baseline": {"Accuracy": 5.3, "Precision": 4.8, "Recall": 3.6},
}

style_map = {
    "Accuracy": '',
    "Precision": r"\\",
    "Recall": 'x'
}

plot("general_bar_plot", data_dict, style_map=style_map, save="bar")
plt.show()
```

Snippet 5. Bar plot with `Swizz`.

B.2. Matplotlib Equivalent with Matching Style

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors
import matplotlib as mpl
import seaborn as sns

# --- Apply Swizz's "latex" theme manually ---
mpl.rcParams.update(mpl.rcParamsDefault)
plt.style.use("seaborn-v0_8-whitegrid")
sns.set_palette("colorblind")
mpl.rcParams.update({
    "font.family": "Times New Roman",
    "font.size": 18,
    "axes.labelsize": 18,
    "axes.titlesize": 18,
    "axes.labelpad": 6.0,
    "axes.labelcolor": "black",
    "axes.linewidth": 1.0,
    "axes.grid": True,
    "axes.grid.axis": "both",
    "axes.spines.top": False,
    "axes.spines.right": False,
    "grid.linestyle": "--",
    "grid.alpha": 0.3,
    "grid.color": "gray",
    "xtick.labelsize": 16,
    "ytick.labelsize": 16,
    "figure.dpi": 300,
    "figure.figsize": (12, 7),
    "figure.facecolor": "white",
    "figure.autolayout": True,
    "lines.linewidth": 2.5,
    "lines.markersize": 6,
    "legend.frameon": True,
    "legend.framealpha": 0.9,
    "legend.edgecolor": "grey",
    "legend.fontsize": 16,
    "legend.handlelength": 1.5,
    "legend.loc": "upper right"
})

# --- Plotting ---
data_dict = {
    "Forward": {"Accuracy": 4.2, "Precision": 3.5, "Recall": 2.1},
    "Reverse": {"Accuracy": 6.0, "Precision": 5.2, "Recall": 4.8},
    "Baseline": {"Accuracy": 5.3, "Precision": 4.8, "Recall": 3.6},
}

fig, ax = plt.subplots(figsize=(12, 7))
categories = list(data_dict.keys())
indices = np.arange(len(categories))
metrics = list(data_dict[next(iter(data_dict))].keys())

bar_width = 0.25
style_map = {"Accuracy": '', "Precision": '\\', "Recall": 'x'}
color_map = {metric: None for metric in metrics}
```

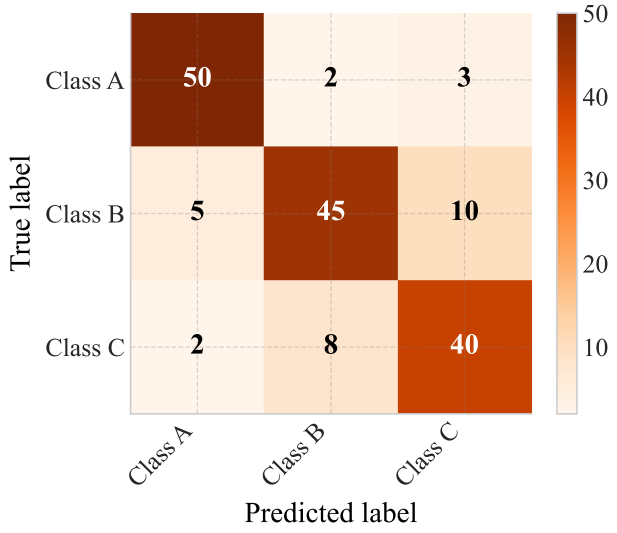
```
bar_positions_list = []
for i, metric in enumerate(metrics):
    values = [data[metric] for data in data_dict.values()]
    positions = indices + (i - len(metrics) / 2) * bar_width
    bars = ax.bar(positions, values, bar_width, label=metric,
                  color=color_map[metric], hatch=style_map[metric], linewidth=1)
    bar_positions_list.append(positions)
    for rect in bars:
        edge = mcolors.to_rgba(rect.get_facecolor(), alpha=1.0)
        edge = mcolors.to_hex([min(1, c * 0.6) for c in edge[:3]])
        rect.set_edgecolor(edge)
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2, height + 0.1,
                f'{height:.1f}', ha='center', va='bottom', color=edge,
                fontweight='bold', fontsize=12)
        ax.plot([rect.get_x() + rect.get_width() / 2]*2, [height, height + 0.1],
                color=edge, lw=1.5)

centers = np.mean(np.array(bar_positions_list), axis=0)
ax.set_xticks(centers)
ax.set_xticklabels(categories)
ax.set_ylabel("Value")
ax.legend(loc="upper right", ncol=len(metrics))
plt.tight_layout()
plt.savefig("bar.png", dpi=300, bbox_inches="tight")
plt.savefig("bar.pdf", dpi=300, bbox_inches="tight")
plt.show()
```

Snippet 6. Bar plot with matplotlib from scratch.

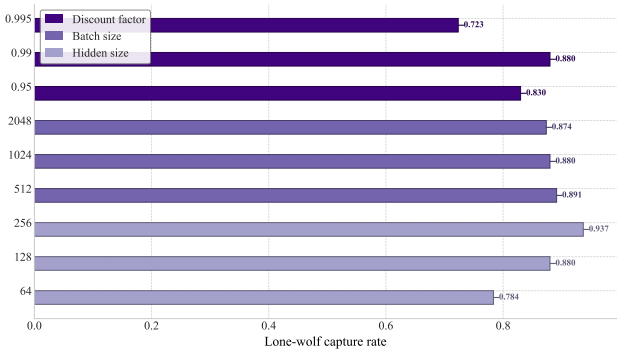
B.3. Summary

`Swizz` condenses over 50 lines of Matplotlib styling and layout code into a single declarative call. While Matplotlib offers full control, it requires manual setup for fonts, styles, and annotations. In contrast, `Swizz` provides named themes and one-liner plot functions to generate consistent, publication-ready figures with minimal effort.

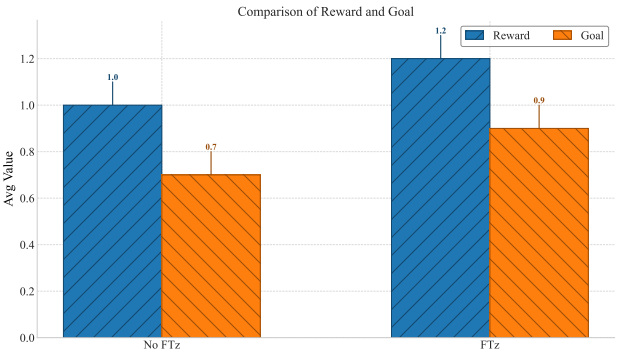
C. Plot examples



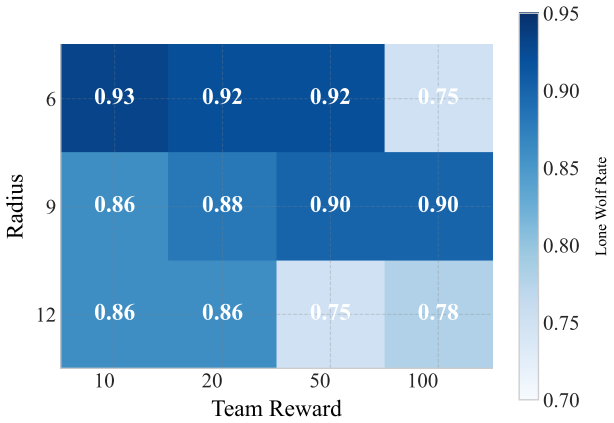
(a) Confusion matrix



(b) General horizontal bar plot



(c) General bar plot



(d) Heatmap

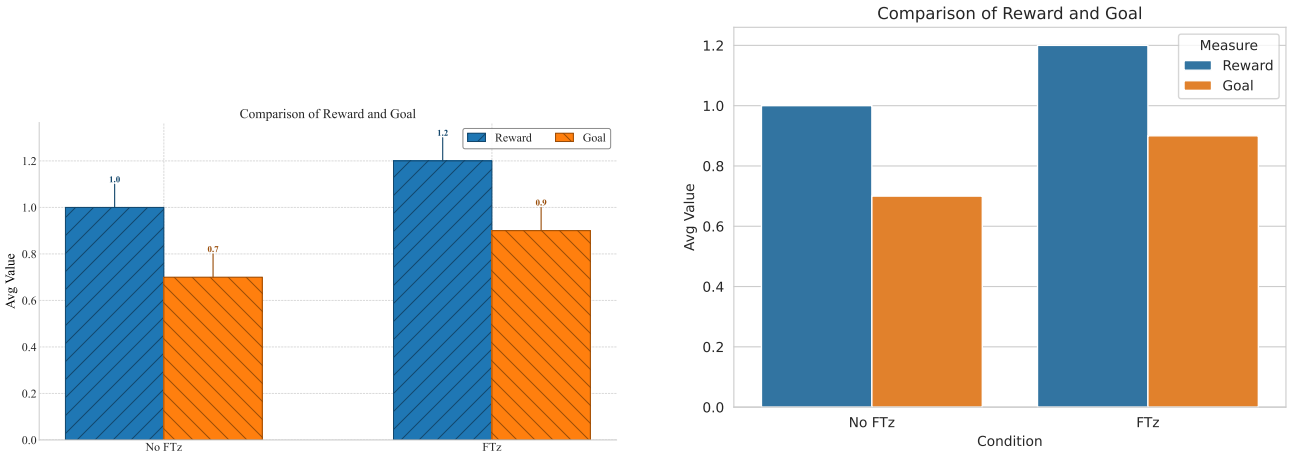
Figure 5. (a) Confusion matrix. (b) General horizontal bar plot. (c) General bar plot. (d) Heatmap.

D. Comparison with existing libraries

Library	Basic one-liner	Fully-annotated one-liner	Grid/ facet	Styling themes	LaTeX figure export	LaTeX table helper	Learning curve
Swizz	✓	✓	✓	✓	✓ (auto)	✓	Low
Seaborn	✓	~	✓	✓	✓*	✗	Low–Mod.
Plotly Express	✓	~	✓	✓	✓*	✗	Low
Altair	✓	~	✓	✓	✓*	✗	Moderate
HoloViews	✓	~	✓	✓	✓*	✗	Steep
tueplots	~	✗	✗	✓	✓*	✗	Low
MatplotX	~	✗	✗	✓	✓*	✗	Low

Table 2. Feature comparison of Swizz with popular high-level Python visualisation libraries. "Fully-annotated one-liner" means the same statement can set titles, axis labels *and* arrange multi-subplots.

*exports static PDF/SVG/PNG; no automatic LaTeX wrapper generated. A tilde (˜) indicates partial or limited support that typically requires follow-up commands.



(a) General bar plot from `Swizz`

(b) Seaborn's `barplot`

Figure 6. Comparison of two implementations of bar plots on the same data.