Matplotlib : Data Visualization

Minor in AI, IIT Ropar

28th Feb, 2025

1 Background

Working as intern for a tech startup, we are helping them visualize the weekly data on code production and bug detection. We want to gain insights into development process and identify potential areas for improvement.

2 Objective

To create an informative and visually appealing representation of the startup's weekly coding and debugging data using Matplotlib, a powerful Python library for data visualization.

3 Implementation

3.1 Step 1: Data Collection and Preparation

The startup provided us with the following data for a week:

- Days of the week: Monday to Sunday
- Lines of code written each day: [50, 120, 200, 180, 300, 50, 20]
- Bugs found each day: [5, 10, 20, 15, 30, 5, 1]

3.2 Step 2: Setting Up the Environment

We begin by importing the necessary libraries:

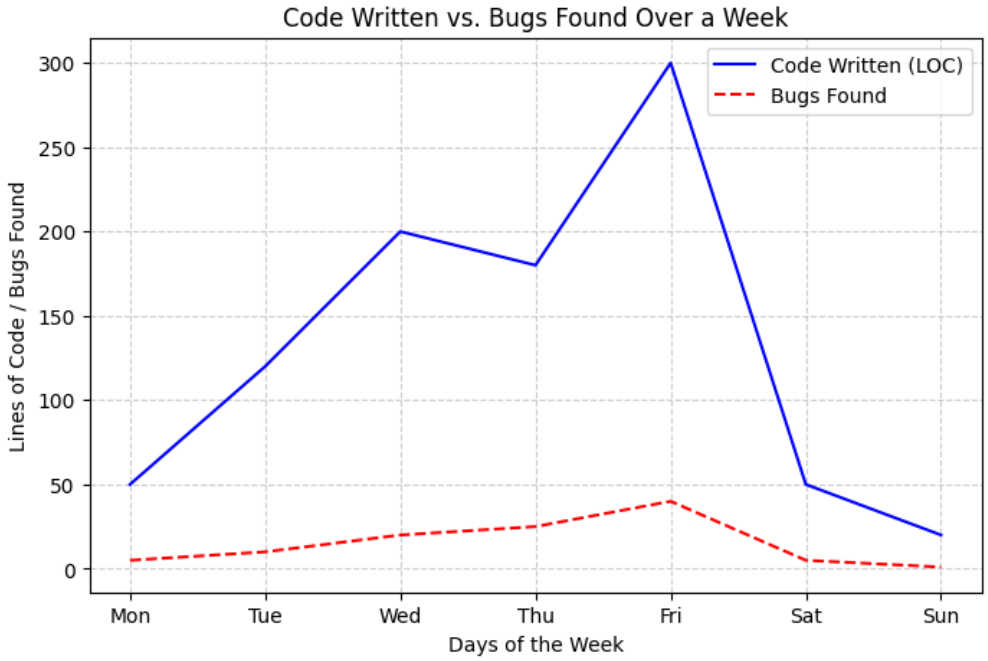
```
import matplotlib.pyplot as plt
```

3.3 Step 3: Creating the Visualization

We use Matplotlib to create a line plot that compares the lines of code written to the bugs found:

```
import matplotlib.pyplot as plt

# Data: Days of the week
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

# Data: Code written (in lines) & Bugs found
code_written = [50, 120, 200, 180, 300, 50, 20]  # Lines of Code
bugs_found = [5, 10, 20, 15, 30, 5, 1]  # Number of bugs

# Create the plot
plt.figure(figsize=(8, 5))
```

```
# Plot Code Written
plt.plot(days, code_written, linestyle='-', color='b', label="Code Written (LOC)")

# Plot Bugs Found
plt.plot(days, bugs_found, linestyle='--', color='r', label="Bugs Found")

# Add labels, title, and legend
plt.xlabel("Days of the Week")
plt.ylabel("Lines of Code / Bugs Found")
plt.title("Code Written vs. Bugs Found Over a Week")
plt.legend()  # Show legend
plt.grid(True, linestyle="--", alpha=0.6)

# Show the plot
plt.show()
```



Upon examining the plot, we observe:

- A peak in code production on Friday (300 lines)
- A corresponding peak in bugs found on Friday (30 bugs)
- Lower code production and bug detection on weekends

3.4 Step 5: Bugs not found

Let's suppose, bugs_found = [5, 10, 20, None, 30, 5, 1]
i.e., for Thurday we didn't had Bugs data then,
To handle the missing data for Thursday:

- We could calculate the mean of the available bug counts, excluding the outlier (Friday's count).
- Alternatively, we could use interpolation based on surrounding days' data.

3.5 Step 6: Saving the Visualization

To preserve the visualization for future reference:

```
plt.savefig('weekly_code_bugs_report.png', dpi=300, bbox_inches='tight')
```

4 Matplotlib Basics

Matplotlib is a powerful Python library for creating static, animated, and interactive visualizations. It provides a MATLAB-like interface for creating plots and figures.

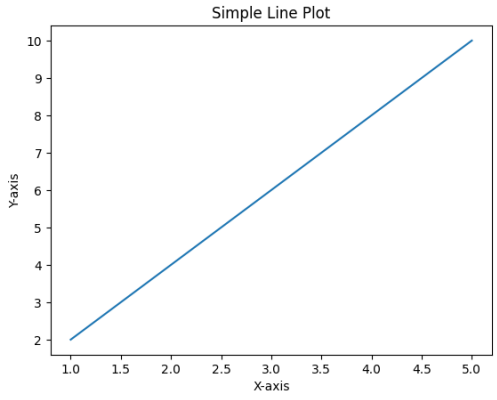
Key features:

- Wide variety of plot types
- High degree of customization
- Integration with NumPy and Pandas
- Support for multiple output formats (PNG, PDF, SVG, etc.)

5 Basic Plotting

The most common way to use Matplotlib is through its pyplot interface:

```
import matplotlib.pyplot as plt

# Create data
x = [1,2,3,4,5]
y = [2,4,6,8,10]

# Create line plot
plt.plot(x, y)

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')

# Display the plot
plt.show()
```



6 Plot Types

Matplotlib supports many types of plots:

- Line plots: `plt.plot(x, y)`
- Scatter plots: `plt.scatter(x, y)`
- Bar plots: `plt.bar(x, y)`
- Histograms: `plt.hist(x)`
- Box plots: `plt.boxplot(x)`
- Pie charts: `plt.pie(x)`

7 Customizing Plots

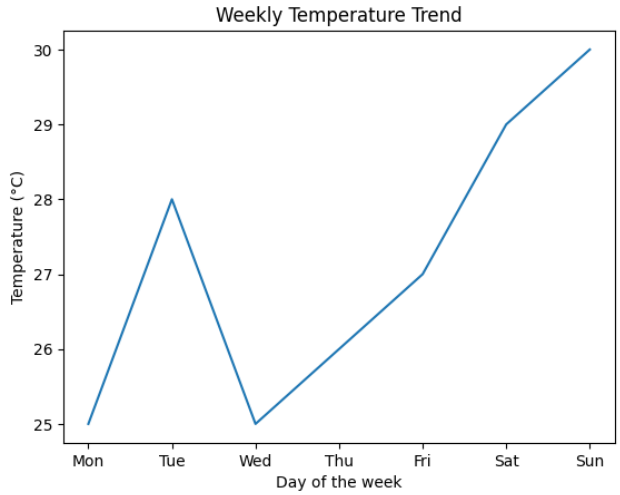
Plots can be customized in various ways:

- Colors: `plt.plot(x, y, color='red')`
- Line styles: `plt.plot(x, y, linestyle='--')`
- Markers: `plt.plot(x, y, marker='o')`
- Axis limits: `plt.xlim(0, 10)`, `plt.ylim(0, 20)`
- Legends: `plt.legend(['Data 1', 'Data 2'])`
- Grid lines: `plt.grid(True)`

8 Weekly Temperature Trend

Take weekly data on temperature of your area from any website and put it in a python list. // We display it using matplotlib and observe the trend.

```
import matplotlib.pyplot as plt

# Sample weather data (replace with your actual data)
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
temperatures = [25, 28, 25, 26, 27, 29, 30]

# Create the line plot
plt.plot(days, temperatures)

# Add labels and title
plt.xlabel("Day of the week")
plt.ylabel("Temperature (°C)")
plt.title("Weekly Temperature Trend")

# Display the plot
plt.show()
```



9 Interactive Plotting

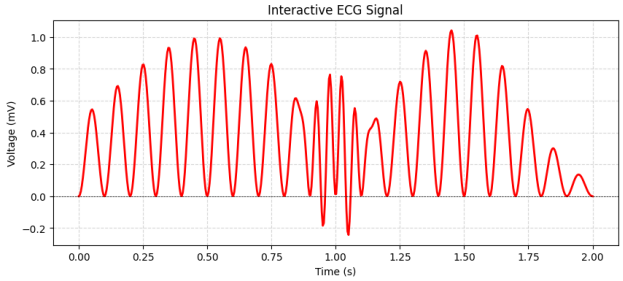
9.1 ECG Signal Graph

Let's suppose you are a researcher studying the topic of heart health and analyzing vast data on ECG signal. Clearly, the ECG signal data in table format will be very bland. Therefore, you can take that data and transform it into a interactive and much easier to understand visualization(just like below one).

Try changing the sliders of Heart Rate and Amplitude.

```
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# Generate x values (time)
x = np.linspace(0, 2, 500)

# Function to create a simulated ECG signal
def ecg_signal(x, heart_rate, amplitude):
    p_wave = np.sin(10 * np.pi * x) * np.exp(-3 * (x - 0.5)**2)  # P wave
    qrs_complex = np.sin(30 * np.pi * x) * np.exp(-100 * (x - 1)**2)  # QRS
    t_wave = np.sin(10 * np.pi * x) * np.exp(-10 * (x - 1.5)**2)  # T wave
    return amplitude * (p_wave + qrs_complex + t_wave) * np.sin(heart_rate * np.pi * x)

# Function to update plot
def update_plot(heart_rate, amplitude):
    y = ecg_signal(x, heart_rate, amplitude)
    plt.figure(figsize=(10, 4))
    plt.plot(x, y, color='red', linewidth=2)

    # Labels and grid
    plt.title("Interactive ECG Signal")
    plt.xlabel("Time (s)")
    plt.ylabel("Voltage (mV)")
    plt.axhline(0, color='black', linewidth=0.5, linestyle="--")
    plt.grid(True, linestyle='--', alpha=0.5)
```

```
    plt.show()

# Create interactive sliders
heart_rate_slider = widgets.FloatSlider(min=5, max=20, step=0.5, value=10, description="Heart Rate")
amplitude_slider = widgets.FloatSlider(min=0.5, max=2.0, step=0.1, value=1.0, description="Amplitude")

# Display interactive plot
display(widgets.interactive(update_plot, heart_rate=heart_rate_slider, amplitude=amplitude_slider))
```



Don't worry you don't need to understand the above code fully, just try to understand how things are working from a top level(like how function is called). Revise sin, cos, exponential functions and how to use them using numpy library.

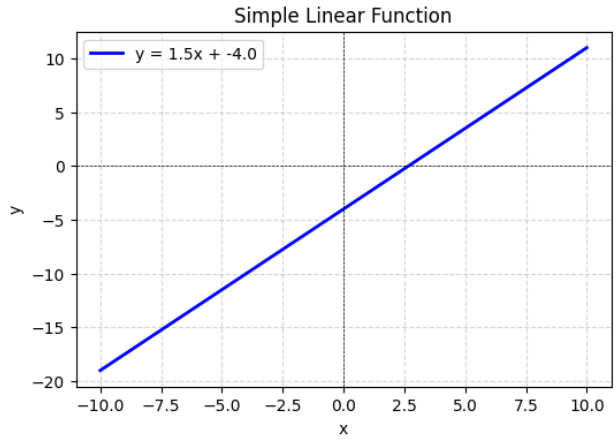
9.2 Simple Linear Function

Earlier we plotted a simple line plot, we can also make it interactive by creating a simple linear function and adding sliders for slope and intercept(if these terms seems foreign to you, revise different equations of 2 D line, one of them is y = mx + c, where m = slope, c = intercept).

Try changing sliders for Slope and Intercept.

```
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# Function to plot a straight line
def plot_line(slope, intercept):
    x = np.linspace(-10, 10, 100)
    y = slope * x + intercept

    plt.figure(figsize=(6, 4))
    plt.plot(x, y, color='blue', linewidth=2, label=f"y = {slope}x + {intercept}")

    # Labels and grid
    plt.title("Simple Linear Function")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.axhline(0, color='black', linewidth=0.5, linestyle="--")
    plt.axvline(0, color='black', linewidth=0.5, linestyle="--")
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)

    plt.show()

# Sliders for slope and intercept
slope_slider = widgets.FloatSlider(min=-5, max=5, step=0.5, value=1, description="Slope")
```

```
intercept_slider = widgets.FloatSlider(min=-10, max=10, step=1, value=0, description="Intercept")

# Display interactive plot
display(widgets.interactive(plot_line, slope=slope_slider, intercept=intercept_slider))
```



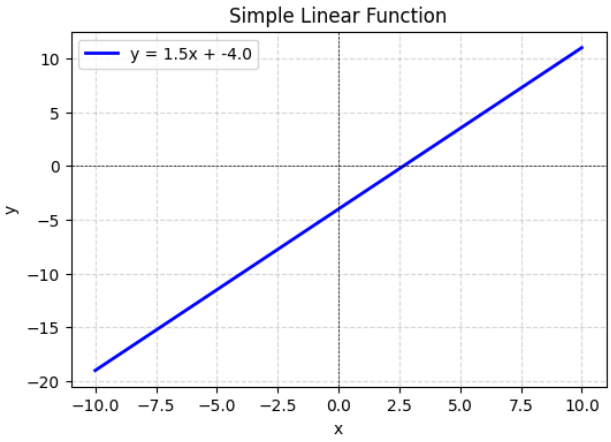
In the above code, try tweaking a few things and understand the plot_line(slope, intercept) function.

9.3 Simple Linear Classifier

Below is a simple linear classifier. Its basically a line plotted in b/w the data points in a way that it forms 2 groups of data (let's say class 0 and class 1). You can adjust the sliders for slope and intercept to see how the data points in class 0 and class 1 gets changed.
This isn't a machine learning model(what's this?? will be covered in later sessions) though, as we are here just segregating the data points. A real classifier would automatically learn the best slope and intercept from the training data.(This will be covered in later sessions too, so don't worry if you don't understand this)

```
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

# Generate random data points
np.random.seed(0)
X = np.random.randn(100, 2)  # 100 points in 2D
y = (X[:, 0] + X[:, 1] > 0).astype(int)  # Simple linear boundary: x + y > 0

# Function to plot decision boundary
def plot_classifier(slope, intercept):
    plt.figure(figsize=(6, 5))

    # Plot data points
    plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='red', label="Class 0")
    plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='blue', label="Class 1")

    # Plot decision boundary
    x_vals = np.linspace(-3, 3, 100)
    y_vals = -slope * x_vals - intercept  # Line equation: y = -mx - b
    plt.plot(x_vals, y_vals, 'green', linewidth=2, label="Decision Boundary")
```

```
    # Labels and grid
    plt.title("Simple Linear Classifier")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.axhline(0, color='black', linewidth=0.5, linestyle="--")
    plt.axvline(0, color='black', linewidth=0.5, linestyle="--")
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)

    plt.show()

# Sliders for slope and intercept of the decision boundary
slope_slider = widgets.FloatSlider(min=-3, max=3, step=0.1, value=1, description="Slope")
intercept_slider = widgets.FloatSlider(min=-3, max=3, step=0.1, value=0, description="Intercept")

# Display interactive plot
display(widgets.interactive(plot_classifier, slope=slope_slider, intercept=intercept_slider))
```

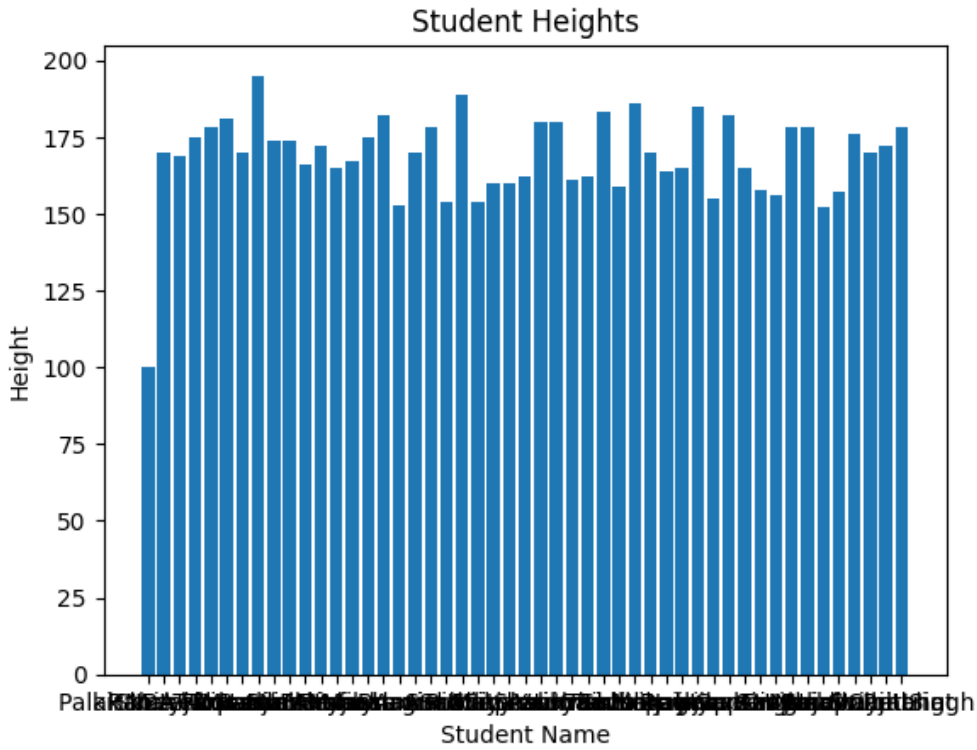


10 Distribution of Student Heights

For visualizing the student heights using matplotlib library, we performed the following steps:-

- Uploaded the student-height.xlsx file to colab(upload option in file icon on left pane).
- when creating pandas dataframe(df), used, pd.read_excel('student-height.xlsx', sheet_name='Sheet1')
- Plotted bar chart using the 2 columns of the dataframe.

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel('student-height.xlsx', sheet_name='Sheet1')
plt.bar(df['Student Name'], df['Height'])
plt.xlabel("Student Name")
plt.ylabel("Height")
plt.title("Student Heights")
plt.show()
```



Some problems with this graph:

- The x labels(student names) are all mushed together
- The heights in this bar chart are for each and every student, maybe there's a better way to visualize it.

We could use normal distribution instead or, even histogram. But it comes at a cost which is that we wouldn't be able to figure out a specific student's height by just looking at the plot in that case.
So, there's always drawbacks and other things we should keep in mind before choosing a plot for best visualization.
This will be covered more in later sessions.

11 Best Practices

- Keep plots simple and focused
- Use appropriate plot types for your data
- Label axes and include titles
- Use color effectively but not excessively
- Consider colorblind-friendly palettes
- Ensure text is readable (font size, contrast)
- Use meaningful scales and units