How to Easily Plot a Normal Distribution in Python Using Matplotlib

Authored by

stats writer

December 6, 2025

Data visualization is an indispensable skill in modern statistical analysis and data science. When dealing with continuous variables, the Normal Distribution, often referred to as the Gaussian distribution, stands out as one of the most fundamental shapes. Plotting this distribution allows us to visually assess data characteristics, understand central tendency, and identify variability. Fortunately, performing this task in Python is highly efficient, primarily utilizing powerful libraries designed specifically for numerical operations and plotting.

This comprehensive guide will walk you through the process of generating and visualizing the bell curve shape of the Normal Distribution using industry-standard Python tools. We rely heavily on the NumPy library for generating numerical sequences and the Matplotlib library, which serves as the backbone for creating high-quality static, animated, and interactive visualizations. Crucially, the statistical heavy lifting--calculating the curve based on parameters--is managed by the robust SciPy library, specifically its `stats` module.

The fundamental methodology involves three distinct steps: defining the range of values (the x-axis), calculating the probability density for each value using the provided mean (mu) and standard deviation ($sigma$), and finally rendering the resulting curve using the Matplotlib plotting function. By the end of this tutorial, you will possess a strong foundational understanding of how these powerful libraries interact to produce accurate statistical plots, enabling you to visualize distributional parameters effortlessly.

Understanding the Normal Distribution Parameters

Before diving into the code, it is essential to appreciate the two key parameters that entirely define any Normal Distribution: the mean (mu) and the standard deviation ($sigma$). The mean dictates the center point of the distribution--it is the peak of the bell curve. A change in the mean shifts the entire curve horizontally along the x-axis without changing its shape.

In contrast, the standard deviation is a measure of the dispersion or spread of the data around the mean. A small standard deviation indicates that the data points tend to be very close to the mean, resulting in a tall, narrow bell curve. Conversely, a large standard deviation signifies that the data points are spread out over a wider range, yielding a shorter, broader curve. Understanding the impact of these two parameters is critical when plotting, as they determine the exact geometry of the resulting graph.

The function we use to generate this curve is the Probability Density Function (PDF). The PDF, specifically the Gaussian PDF provided by the SciPy library's `norm` method, calculates the relative likelihood of a continuous random variable taking on a given value. It is crucial to remember that the area under the entire PDF curve must always integrate to **1**, representing 100% probability, regardless of the values chosen for mu and $sigma$.

Essential Python Libraries for Visualization

To successfully plot statistical distributions, we require three core libraries in the Python data science ecosystem: NumPy, Matplotlib, and SciPy. Each library plays a specific, interdependent role in the plotting process. We initiate the process by importing these packages, typically using the standardized aliases `np` for NumPy and `plt` for Matplotlib's `pyplot` submodule.

The NumPy library (Numerical Python) is utilized to efficiently create the domain for our plot--the range of x-values. This is achieved using the `np.arange()` function, which generates evenly spaced values within a specified interval. Since distributions are continuous, we require a fine granularity (small step size) to ensure the plotted curve appears smooth rather than jagged. This array of x-values forms the input dataset for calculating the density values.

The statistical heavy lifting is handled by the `norm` function found within `scipy.stats`. This function, which implements the Gaussian Probability Density Function (PDF), accepts the array of x-values along with the required mu (mean) and $sigma$ (standard deviation) arguments. The output is a corresponding array of y-values, representing the density or height of the curve at each x-point. Finally, Matplotlib takes these two arrays (x and y) and translates them into a graphical representation using the `plt.plot()` function.

To plot a standard Normal Distribution in Python, which has a mean of 0 and a standard deviation of 1, you can use the following fundamental syntax:

#x-axis ranges from -3 and 3 with .001 steps
x = np.arange(-3, 3, 0.001)

#plot normal distribution with mean 0 and standard deviation 1
plt.plot(x, norm.pdf(x, 0, 1))

The **x** array, created using NumPy's `arange` function, defines the continuous domain for the x-axis. Subsequently, the **plt.plot()** function is invoked, utilizing the `norm.pdf(x, 0, 1)` calculation to produce the density curve for the Normal Distribution with the specified mean of 0 and standard deviation of 1.

The following practical examples demonstrate how to incorporate these functions into complete, runnable Python scripts.

Step-by-Step Implementation: Plotting a Single Distribution (Example 1)

Our first example focuses on plotting the standard Normal Distribution, which is characterized by a

mean (mu) equal to 0 and a standard deviation ($sigma$) equal to 1. This is the simplest and most commonly referenced form of the Gaussian curve. To execute this plot, we must first ensure all necessary libraries--NumPy, Matplotlib's `pyplot`, and `norm` from SciPy--are imported into our environment.

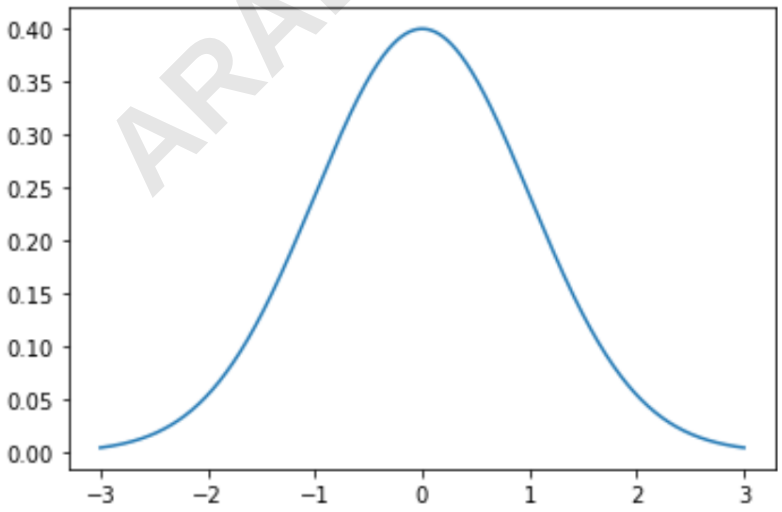
After imports, the critical step is defining the input range. We use `np.arange(-3, 3, 0.001)` to generate the x-values. The range from **-3 to 3** covers approximately **99.7%** of the data in a standard normal curve (the empirical rule), making it an ideal visual boundary. The step size of **0.001** ensures maximum smoothness for the continuous curve. Once the domain is defined, we call `plt.plot()`, passing the x-array and the corresponding density calculation `norm.pdf(x, 0, 1)` as arguments.

The code below illustrates the complete script required to generate this foundational plot. Notice how the structure clearly separates the data generation (using NumPy) from the statistical calculation (SciPy) and the final visualization (Matplotlib). This modular approach is typical in scientific Python programming.

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

#x-axis ranges from -3 and 3 with .001 steps
x = np.arange(-3, 3, 0.001)

#plot normal distribution with mean 0 and standard deviation 1
plt.plot(x, norm.pdf(x, 0, 1))



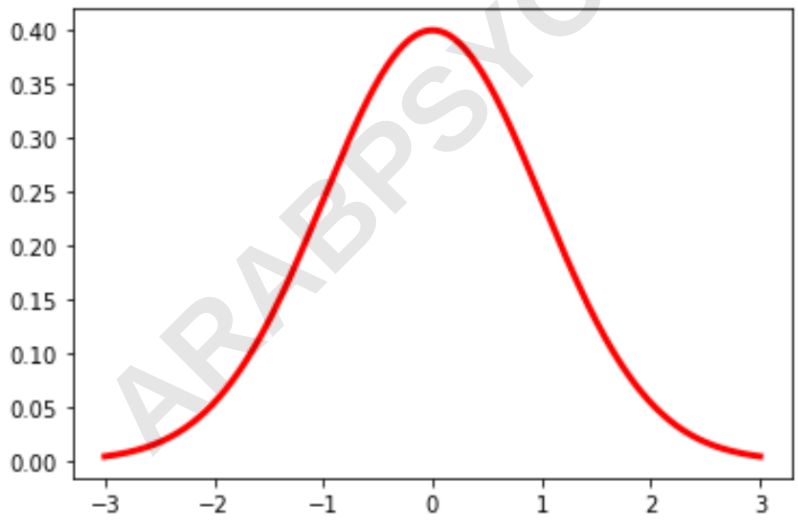
Customizing the Visualization Aesthetics

While the basic plot is statistically sound, customizing the aesthetics greatly enhances readability and professionalism. The Matplotlib `plt.plot()` function accepts numerous optional arguments that allow fine-grained control over the appearance of the generated line. Two of the most commonly adjusted parameters are `color`, which sets the hue of the line, and `linewidth`, which controls its thickness.

By default, Matplotlib uses its cyclical color scheme, but explicitly defining the color (e.g., 'red', 'blue', or a hex code) ensures consistency across different visualizations. Similarly, increasing the `linewidth` makes the curve more pronounced and easier to view, especially in presentations or printed materials. These customizations are applied directly within the `plt.plot()` call, making the code concise and efficient.

In the following demonstration, we modify the standard plot to use a vibrant **red** color and increase the line width to **3 units**. This simple modification immediately draws attention to the curve and distinguishes it from default plots. It is highly recommended to leverage these aesthetic controls when creating complex graphs featuring multiple distributions or datasets.

plt.plot(x, norm.pdf(x, 0, 1), color='red', linewidth=3)



Visualizing the Impact of Parameters (Example 2)

One of the most powerful uses of visualization is comparing statistical models directly. By plotting multiple Normal Distribution curves on the same axes, we can immediately grasp the visual effect of changing the mean (mu) or the standard deviation ($sigma$). For instance, keeping the mean

constant at 0 while increasing the standard deviation demonstrates how the probability mass spreads out, leading to flatter curves.

To implement this, we simply call the `plt.plot()` function multiple times, once for each distribution we wish to display. Each call uses the same x-array (the domain) but specifies unique mu and $sigma$ parameters within the `norm.pdf()` calculation. It is absolutely crucial to include the `label` argument in each plot call. This label is used later by the `plt.legend()` function to identify which line corresponds to which parameter set, ensuring the chart is fully interpretable.

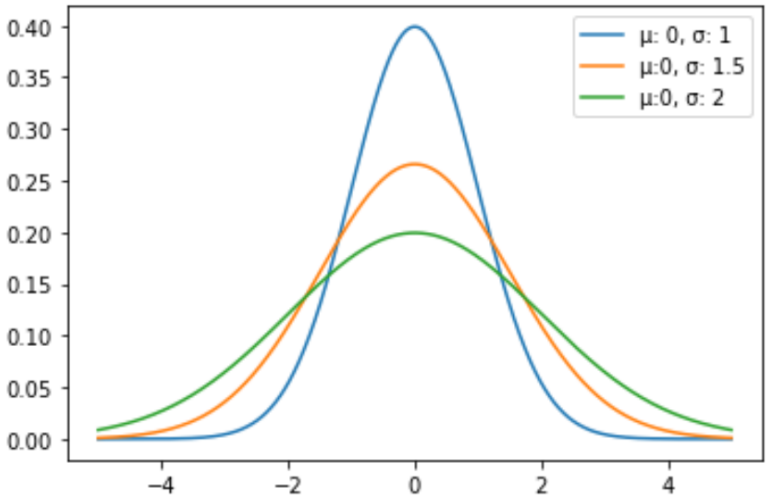
In this example, we maintain a mean of 0 across all distributions but vary the standard deviation from 1 to 2. We also expand the x-axis range from **-5 to 5** to accommodate the increased spread of the data, thereby ensuring that the full extent of the flatter curves is visible. The final step involves calling `plt.legend()` to display the labels defined for each curve.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

#x-axis ranges from -5 and 5 with .001 steps
x = np.arange(-5, 5, 0.001)

#define multiple normal distributions
plt.plot(x, norm.pdf(x, 0, 1), label='μ: 0, σ: 1')
plt.plot(x, norm.pdf(x, 0, 1.5), label='μ:0, σ: 1.5')
plt.plot(x, norm.pdf(x, 0, 2), label='μ:0, σ: 2')

#add legend to plot
plt.legend()
```



Enhancing Readability with Titles and Labels

A professional visualization must be self-explanatory. This means adding descriptive titles and axis labels is mandatory, particularly when presenting data to stakeholders or integrating figures into reports. Matplotlib provides straightforward functions--`plt.title()`, `plt.xlabel()`, and `plt.ylabel()`--to annotate the graph clearly. Furthermore, we can refine the legend display to be more informative.

When working with multiple lines, as in Example 2, specifying individual colors is often necessary to distinguish the curves effectively, even if the legend is present. We can reuse the customization techniques from Example 1, applying unique colors to each `plt.plot()` call. Additionally, we use the `title` parameter within `plt.legend()` to provide context for the parameters shown (e.g., 'Parameters').

The following expanded code block demonstrates how to combine all visualization best practices: defining distinct line colors, adding a descriptive chart title, labeling both the x and y axes (representing the variable value and the Probability Density Function, respectively), and formalizing the legend structure. This results in a fully polished and easily digestible figure.

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

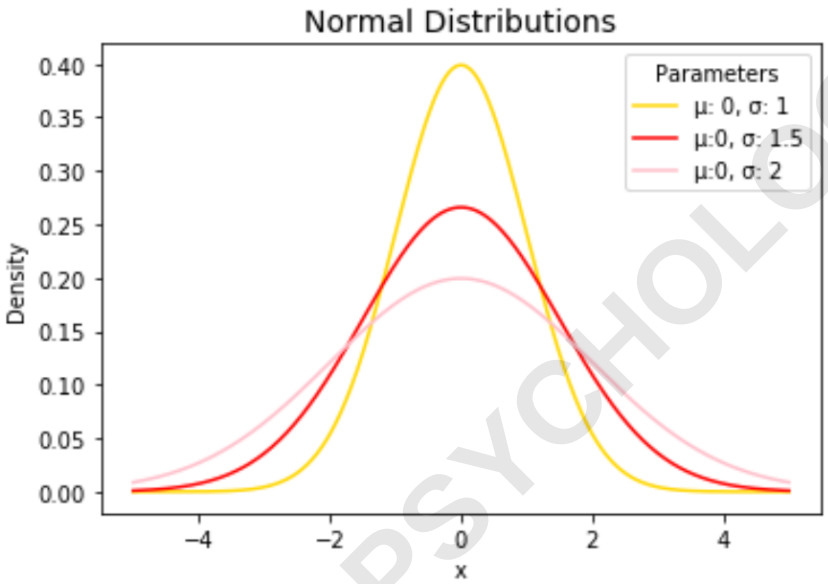
#x-axis ranges from -5 and 5 with .001 steps
x = np.arange(-5, 5, 0.001)

#define multiple normal distributions

```
plt.plot(x, norm.pdf(x, 0, 1), label='μ: 0, σ: 1', color='gold')
plt.plot(x, norm.pdf(x, 0, 1.5), label='μ:0, σ: 1.5', color='red')
plt.plot(x, norm.pdf(x, 0, 2), label='μ:0, σ: 2', color='pink')

#add legend to plot
plt.legend(title='Parameters')

#add axes labels and a title
plt.ylabel('Density')
plt.xlabel('x')
plt.title('Normal Distributions', fontsize=14)
```



Summary of Key Functions and Resources

Successful plotting of the Normal Distribution relies on the seamless cooperation of specific functions from the Python ecosystem. We utilized `numpy.arange()` to define the continuous domain of the plot, ensuring fine granularity for visual smoothness. For the core statistical calculation, `scipy.stats.norm.pdf(x, mu, sigma)` was the critical component, translating the theoretical distribution parameters into plottable density values.

The visualization itself was driven entirely by Matplotlib, employing several key functions for rendering and annotation:

`plt.plot(x, y, ...)`: The primary function used to draw the line graph, accepting coordinates (x and y) and optional styling arguments (e.g., **color**, **linewidth**, **label**).

`plt.legend()`: Essential for identifying multiple curves on the same plot, drawing information from the `label` arguments specified in `plt.plot()`.
`plt.title()`, `plt.xlabel()`, `plt.ylabel()`: Used for adding necessary context to the chart, fulfilling the requirements for professional data reporting.

For advanced customization and exploration of all available plotting options, refer to the official Matplotlib documentation for an in-depth explanation of the **plt.plot()** function and its extensive parameter list. Mastering these tools ensures you can accurately and aesthetically represent any continuous probability distribution.