How to Easily Plot a Chi-Square Distribution in Python

Authored by

stats writer

December 6, 2025

Visualizing the Chi-Square Distribution is a fundamental task in statistics, particularly when performing hypothesis testing, analyzing contingency tables, or determining goodness-of-fit. Python offers powerful, efficient tools for this purpose, primarily through the integration of the numerical library SciPy and the visualization tool Matplotlib. By leveraging these robust packages, data scientists and researchers can generate clean, informative plots of this critical probability distribution.

The core of plotting the distribution relies on SciPy's statistical module (`scipy.stats`), which provides comprehensive functionality for various statistical distributions. Specifically, the `chi2` object within this module contains essential methods for computing the distribution's properties, such as the probability density and cumulative values. Understanding how these functions--`chi2.pdf()`, `chi2.cdf()`, and `chi2.ppf()`--interact with Matplotlib is key to mastering Chi-Square visualization in Python.

This guide provides a detailed, step-by-step approach to plotting both single and multiple Chi-Square Distribution curves, including methods for enhancing the aesthetic quality of your statistical graphs for professional presentation.

Introduction to Chi-Square Plotting in Python

The process of plotting any continuous distribution in Python requires defining a range of x-values and then calculating the corresponding y-values (the probability density) for each point. For the Chi-Square Distribution, the shape is governed solely by one parameter: the degrees of freedom (df). Since the Chi-Square test statistic is always non-negative, the x-axis for the plot typically begins at zero.

The implementation hinges on the specialized functions within the `scipy.stats.chi2` object. For visualization purposes, we primarily rely on the `.pdf()` method. The probability density function (PDF) defines the curve we traditionally associate with the distribution, showing the density of the probability mass across the domain of possible values. By feeding a sequence of x-values and a specific degree of freedom parameter into `chi2.pdf()`, we obtain the data points necessary for plotting.

While `chi2.pdf()` is used for the distribution curve itself, it is beneficial to be aware of related functions. The `chi2.cdf()` function calculates the Cumulative Distribution Function, which is the area under the PDF curve up to a given x-value, providing the cumulative probability. Furthermore, `chi2.ppf()` (Percent Point Function) calculates the critical values for a given probability, often utilized when setting significance levels for statistical tests. These functions together form a powerful suite for comprehensive statistical analysis within Python.

Essential Python Libraries for Statistical Visualization

Generating high-quality statistical plots requires the seamless integration of three primary Python libraries: NumPy, SciPy, and Matplotlib. Each library plays a distinct and critical role in the visualization pipeline, ensuring mathematical accuracy and graphical fidelity.

NumPy, or Numerical Python, is fundamental. It provides the core numerical array structures necessary for handling large datasets efficiently. In plotting, NumPy is essential for defining the domain of the plot--the x-axis values. We typically use functions like `np.arange()` or `np.linspace()` to create an array of evenly spaced points across the desired range (e.g., 0 to 20). Using small steps (like 0.001) ensures the resulting plot is a smooth, continuous curve, accurately reflecting the distribution's theoretical shape.

SciPy is the specialized statistical library. It takes the array of x-values provided by NumPy and uses the `chi2.pdf()` method to calculate the corresponding probability density for each point, given a fixed degrees of freedom parameter. The output is a new NumPy array containing the calculated y-values, which represent the height of the curve at every x-coordinate.

Finally, Matplotlib, specifically the `matplotlib.pyplot` module, is responsible for rendering the graph. It accepts the paired x and y arrays and uses the `plt.plot()` function to connect these points, thereby drawing the visualized distribution curve. Matplotlib also provides extensive customization options for titles, labels, colors, and line styles, ensuring the final output is suitable for publication or detailed analysis.

Setting Up the Environment and Basic Plotting Syntax

Before executing any plotting commands, it is crucial to ensure that the necessary libraries are imported and correctly aliased. Standard practice involves importing NumPy as `np`, Matplotlib's plotting submodule as `plt`, and the specific distribution object, `chi2`, from `scipy.stats`.

The degrees of freedom (df) is the single most important parameter when working with the Chi-Square Distribution. This value dictates the mean and variance of the distribution, fundamentally shaping its skewness and peak. For smaller degrees of freedom (e.g., df < 5), the distribution is highly skewed to the right. As the degrees of freedom increase, the distribution becomes more symmetrical and bell-shaped, gradually approaching a normal distribution.

To plot a Chi-Square Distribution in Python, you can use the following syntax, which involves defining the x-range and then calling the plotting function:

To plot a Chi-Square Distribution in Python, you can use the following syntax:

#x-axis ranges from 0 to 20 with .001 steps

```
x = np.arange(0, 20, 0.001)

#plot Chi-square distribution with 4 degrees of freedom
plt.plot(x, chi2.pdf(x, df=4))
```

The **x** array, generated using NumPy's `arange` function, establishes the range for the x-axis, covering values from 0 to 20 in very small increments. The `plt.plot()` function then utilizes this range alongside the calculated probability density function values derived from `chi2.pdf()` to produce the smooth curve for the specified degrees of freedom.

The subsequent examples demonstrate these powerful functions in action, starting with the simplest case: plotting a single distribution.

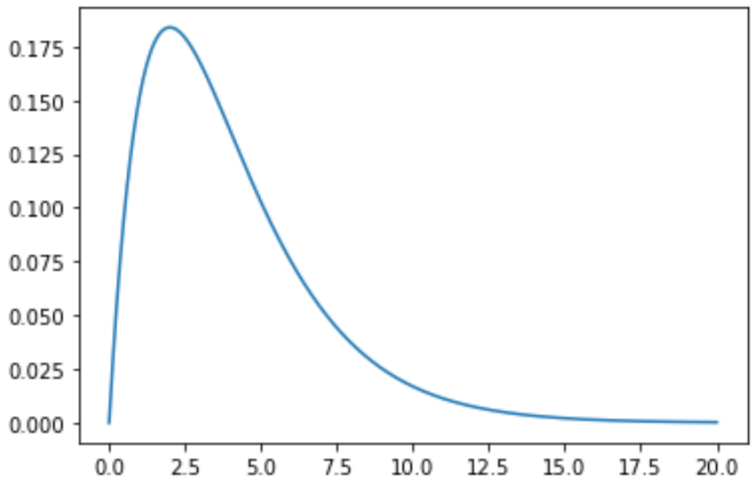
Example 1: Visualizing a Single Chi-Square Distribution Curve

Our first practical example focuses on generating a high-resolution plot for a single instance of the Chi-Square Distribution, specifically using 4 degrees of freedom. This requires importing all components and carefully defining the domain (x-axis) that encapsulates the majority of the probability mass. Since the mean of the Chi-Square distribution is equal to its degrees of freedom, setting the upper bound of the x-axis to about five times the degrees of freedom (20, in this case) is usually sufficient to capture the curve's tail.

The following code snippet combines the necessary imports, array generation, and the core plotting command:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import chi2

#x-axis ranges from 0 to 20 with .001 steps
x = np.arange(0, 20, 0.001)

#plot Chi-square distribution with 4 degrees of freedom
plt.plot(x, chi2.pdf(x, df=4))
```

Executing this code will produce the initial visual representation of the distribution. Notice how the curve begins at zero, rapidly peaks shortly after, and then trails off toward the right. This pronounced right-skew is characteristic of the Chi-Square Distribution when the degrees of freedom value is small.



Customizing Plot Appearance and Aesthetics

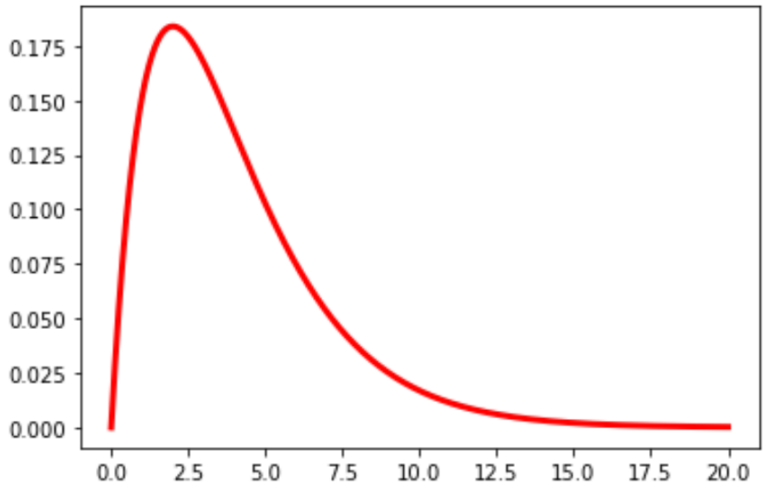
While the basic plot is mathematically correct, enhancing its visual attributes makes it significantly easier to interpret and integrate into reports or presentations. Matplotlib offers direct parameters within the `plt.plot()` function to customize line color, thickness, and style.

Two fundamental customization parameters are `color` and `linewidth`. The `color` parameter accepts standard color names (like 'red', 'blue', 'gold') or hexadecimal color codes. The `linewidth` parameter controls the thickness of the line, measured in points, with the default being 1. By increasing the line width and selecting a distinct color, we can make the distribution stand out more clearly, especially when overlapping multiple curves.

The following modification shows how to specify the line color as red and increase its thickness to 3 points:

plt.plot(x, chi2.pdf(x, df=4), color='red', linewidth=3)

This simple adjustment transforms the visual impact of the distribution plot, improving readability and drawing attention to the curve itself. Proper customization is a crucial step in translating raw statistical output into meaningful data graphics.



Example 2: Comparing Multiple Chi-Square Distributions

One of the most insightful visualizations is the comparison of multiple Chi-Square distributions with varying degrees of freedom (df). This visual comparison clearly illustrates how the distribution's shape evolves as the df parameter increases. Specifically, as df rises, the peak of the probability density function shifts to the right, and the curve becomes less skewed, demonstrating the approach toward symmetry.

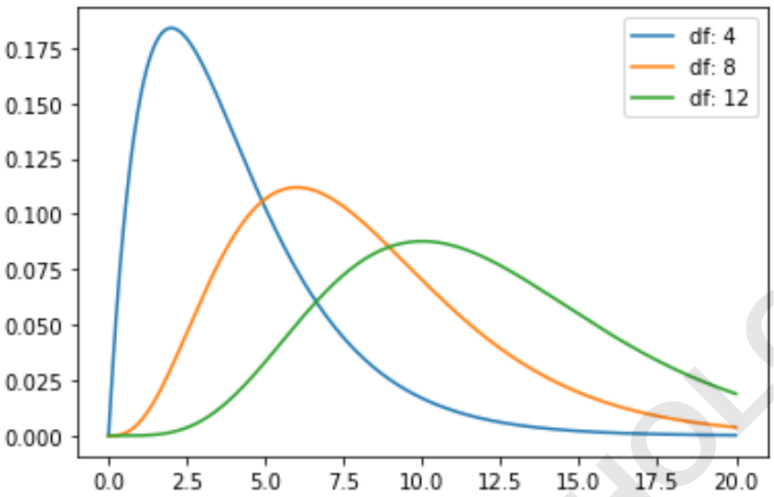
To plot multiple distributions on the same axes, we simply call `plt.plot()` multiple times, once for each degree of freedom we wish to include. It is essential to assign a unique `label` to each plot call; this label is necessary for Matplotlib to automatically generate a descriptive legend, which links each curve to its corresponding parameter value.

The code below plots three distributions with df=4, df=8, and df=12, and then uses `plt.legend()` to display the labels:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import chi2

#x-axis ranges from 0 to 20 with .001 steps
x = np.arange(0, 20, 0.001)

#define multiple Chi-square distributions
plt.plot(x, chi2.pdf(x, df=4), label='df: 4')
plt.plot(x, chi2.pdf(x, df=8), label='df: 8')
plt.plot(x, chi2.pdf(x, df=12), label='df: 12')
```

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

The resulting plot clearly shows the differences in distribution shape. The df=4 curve is highly peaked and skewed, while the df=12 curve is lower, broader, and shows much less skewness, illustrating the strong influence of the degrees of freedom parameter on the underlying statistical model.



Enhancing Multi-Distribution Plots with Labels and Titles

To finalize any statistical plot for professional use, it must be fully annotated with appropriate titles and axis labels. These additions provide context and ensure that the audience immediately understands what the axes represent and the overall focus of the visualization. Matplotlib provides dedicated functions for labeling the x-axis, the y-axis, and the overall chart.

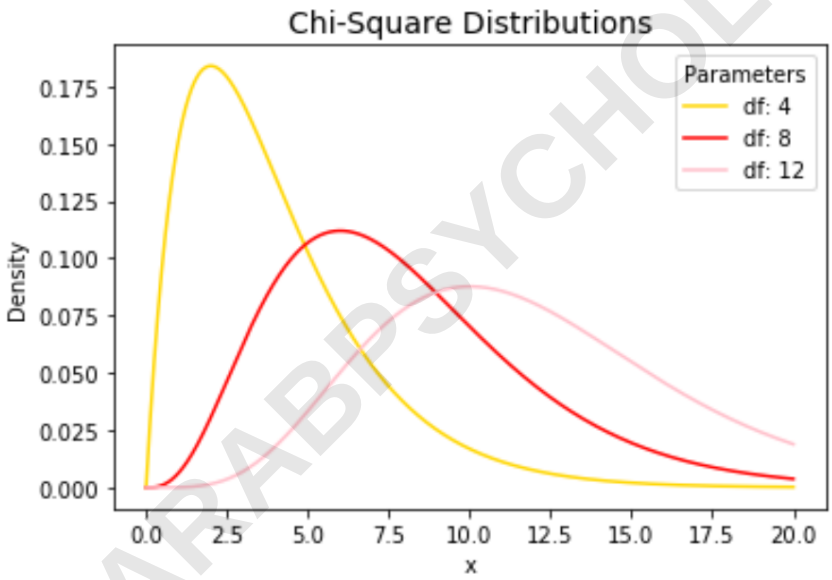
We can further enhance the comparison plot by assigning specific colors to each curve to improve contrast and by adding labels for "Density" (y-axis), "x" (x-axis), and a main title, "Chi-Square Distributions". We also customize the legend by providing it with a title ('Parameters') for clarity.

Here is the complete code, incorporating customization for colors, legend, labels, and title:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import chi2

#x-axis ranges from 0 to 20 with .001 steps
x = np.arange(0, 20, 0.001)
```

```
#define multiple Chi-square distributions
plt.plot(x, chi2.pdf(x, df=4), label='df: 4', color='gold')
plt.plot(x, chi2.pdf(x, df=8), label='df: 8', color='red')
plt.plot(x, chi2.pdf(x, df=12), label='df: 12', color='pink')

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

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

This comprehensive approach results in a fully annotated and visually appealing statistical chart, ready for incorporation into any analytical report. The labels confirm that the y-axis represents the probability density and the x-axis represents the possible values of the Chi-Square statistic.



Summary of Key Plotting Components

Plotting the Chi-Square Distribution effectively in Python relies on a core set of functions from the three major libraries. Mastering these functions allows for generating accurate and customizable statistical graphics.

The essential components covered in this tutorial include:

`np.arange(start, stop, step)`: Used to create the array **`x`** of evenly spaced values, defining the domain (x-axis) of the plot. Using small step values ensures curve smoothness.
`chi2.pdf(x, df)`: The central function, provided by SciPy, which calculates the probability density function values corresponding to the input x-array and the specified degrees of freedom.
`plt.plot(x, y, parameters)`: The primary Matplotlib function used to render the curve. It accepts the x-values and the calculated PDF (y-values), along with optional arguments like `color`, `linewidth`, and `label`.
`plt.legend()`: Displays the legend based on the `label` arguments provided in the corresponding `plt.plot()` calls, crucial for distinguishing multiple distributions.
`plt.xlabel()`, `plt.ylabel()`, `plt.title()`: Functions used to annotate the graph, providing necessary context and readability for the visualization.

For more detailed information regarding the advanced features and customization options available within the `plt.plot()` function, including various line styles, marker options, and handling subplots, specialized documentation is recommended. Refer to the official Matplotlib documentation for an in-depth explanation of visualization techniques.