How to Easily Calculate Manhattan Distance in Python

Authored by

stats writer

December 5, 2025

The calculation of distance metrics is fundamental in fields ranging from computational geometry to advanced data science. Among these metrics, the Manhattan Distance--often referred to as the taxicab distance or city block distance--holds unique significance, particularly when movement is constrained to orthogonal directions. This metric derives its name from the grid layout typical of Manhattan, New York, where travel between two points must follow the street grid rather than a direct straight line.

The core concept of Manhattan Distance involves measuring the distance between two points in a space defined by Cartesian coordinates. Unlike the Euclidean distance, which measures the shortest path (a straight line), the Manhattan Distance is calculated by summing the horizontal and vertical distances. This calculation involves taking the sum of the absolute differences of the coordinates along each axis. It accurately models scenarios where diagonal movement is impossible or irrelevant, making it a critical tool in graph theory, network routing, and various optimization problems.

For practitioners working in machine learning and statistical analysis using Python, implementing this distance calculation efficiently is essential. While a custom function can be written manually, the powerful capabilities of established scientific libraries, such as SciPy, offer optimized and reliable solutions. This comprehensive guide details both the theoretical foundations of the Manhattan Distance and provides practical, verified examples for its calculation in Python, ensuring clear understanding for developers and data scientists alike.

The Mathematical Definition of Manhattan Distance

Understanding the mathematical foundation is crucial before diving into the implementation. The Manhattan Distance is a specific case of the Minkowski distance metric, defined when $p=1$. It provides a clear metric for dissimilarity between two data points or vectors, A and B, within an n-dimensional space.

The mathematical representation for calculating the Manhattan Distance between two vectors, A and B, where $A = (A_1, A_2, dots, A_n)$ and $B = (B_1, B_2, dots, B_n)$, is given by the formula:

Σ|Ai - Bi|

where i iterates from 1 to n, representing the ith element in each vector. This summation emphasizes the cumulative impact of coordinate differences along each axis, ensuring that the resulting distance reflects the path length constrained to the axes of the coordinate system. The use of the absolute value ensures that the distance is always non-negative, regardless of the order of subtraction.

This distance metric is highly effective for measuring the disparity between two feature sets and is widely utilized in various machine learning algorithms, including K-Nearest Neighbors (KNN) when feature dimensions are high or when the data distribution justifies an L_1 norm approach over the traditional L_2 (Euclidean) norm.

Manhattan Distance vs. Euclidean Distance

While the Euclidean distance is arguably the most common distance metric, understanding when and why to select the Manhattan Distance is key to proper modeling. Euclidean distance, calculated as the square root of the sum of the squared differences, represents the direct, shortest path between two points. This is appropriate when movement can occur freely in any direction.

The Manhattan Distance, conversely, is preferred when the path must be constrained to a grid or when feature importance should be treated linearly rather than quadratically. In high-dimensional spaces, Manhattan Distance tends to be more robust to noise and outliers compared to Euclidean distance because it avoids squaring the differences, which can amplify the impact of large deviations. Furthermore, when dealing with certain types of data, such as sparse data or features measured on differing scales, the linear nature of the L_1 norm (Manhattan Distance) often yields more intuitive and useful results than the L_2 norm (Euclidean Distance).

Choosing the correct distance metric is a critical step in algorithm design. If the application involves analyzing feature differences where the sum of individual differences provides a meaningful interpretation of total variance--such as in urban planning, digital image analysis, or certain classification tasks--the Manhattan Distance is often the optimal choice. This tutorial demonstrates two highly effective methods for calculating this essential metric within the Python ecosystem.

Method 1: Writing a Custom Python Function

For quick calculations, educational purposes, or situations where external library dependencies must be minimized, writing a custom function in Python to calculate the Manhattan Distance is straightforward and efficient. This method allows the developer maximum control over the implementation and ensures a deep understanding of the underlying mathematical process.

The following code snippet demonstrates how to define a function, which we name `manhattan`, that takes two vectors as input and iterates through their elements using Python's built-in `zip` function. The function then calculates the absolute difference between corresponding elements and aggregates these differences using the `sum` function, directly implementing the defined mathematical formula.

from math import sqrt

```
#create function to calculate Manhattan distance
def manhattan(a, b):
return sum(abs(val1-val2) for val1, val2 in zip(a,b))

#define vectors
A =
B =

#calculate Manhattan distance between vectors
manhattan(A, B)

9
```

After defining the two sample vectors, `A` and `B`, the execution of the custom function yields a result of **9**. This result confirms the successful calculation of the distance based purely on standard Python operations, demonstrating the power of Python's list comprehensions and iterative capabilities for concise code generation.

Verifying the Custom Function Calculation

To ensure the accuracy of the custom function and reinforce the understanding of the mathematical formula, it is beneficial to perform the calculation manually using the provided sample vectors. This step-by-step verification confirms that the Python code correctly translates the mathematical definition of the Manhattan Distance.

Given the vectors $A = $ and $B = $, we calculate the sum of the absolute differences for each corresponding element pair:

Σ|Ai - Bi| = |2-5| + |4-5| + |4-7| + |6-8|

First, we calculate the absolute difference for each dimension:

|2-5| = |-3| = 3

|4-5| = |-1| = 1

|4-7| = |-3| = 3

|6-8| = |-2| = 2

Finally, we sum these individual absolute differences: $3 + 1 + 3 + 2 = mathbf{9}$. This manual calculation confirms that the Manhattan distance between these two vectors is indeed **9**, validating

the output of the custom Python function.

Method 2: Utilizing the SciPy Library for Optimized Calculation

While custom functions are valuable, real-world data science applications often rely on highly optimized libraries that ensure computational efficiency and numerical stability. The SciPy library, a cornerstone of the scientific Python stack, provides comprehensive tools for mathematical operations, including distance calculations.

SciPy's `spatial.distance` module includes a specialized function for the Manhattan Distance, conveniently named `cityblock()`. This function is highly optimized for performance and should be the preferred method when working with large datasets or integrating into complex computational pipelines. Utilizing established libraries reduces the potential for coding errors and leverages years of community optimization efforts.

The following example demonstrates how to import and use the `cityblock()` function to calculate the Manhattan Distance between the same pair of vectors used in the previous section. Note the clean and concise nature of the library-based implementation.

from scipy.spatial.distance import cityblock

#define vectors
A =
B =

#calculate Manhattan distance between vectors
cityblock(A, B)

9

As expected, the `cityblock()` function returns the identical distance of **9**, confirming its equivalence to the manual calculation and the custom function, while providing a standardized, high-performance interface for metric computation.

Advanced Application: Calculating Distance in Pandas DataFrames

In data analysis and machine learning workflows, data is most frequently structured within Pandas DataFrames. A significant advantage of using the SciPy implementation is its seamless integration with other Python data structures, including Pandas Series (representing columns within a DataFrame). This enables analysts to easily calculate the distance between feature vectors represented as columns.

The ability to calculate distances directly between DataFrame columns is crucial for tasks like feature similarity assessment, clustering, and anomaly detection. In this scenario, each column represents a distinct vector of observations across different data points. The Manhattan Distance calculation provides a metric for how similar or dissimilar two sets of features are, based on the cumulative linear differences.

The following example illustrates how to define a Pandas DataFrame and subsequently use the `cityblock()` function to find the Manhattan Distance between columns `A` and `B`, demonstrating a common pattern in data preprocessing:

from scipy.spatial.distance import cityblock
import pandas as pd

#define DataFrame
df = pd.DataFrame({'A': ,
'B': ,
'C': })

#calculate Manhattan distance between columns A and B
cityblock(df.A, df.B)

9

By passing the Pandas Series objects (`df.A` and `df.B`) directly into the `cityblock()` function, we leverage the vectorized operations inherent in SciPy, resulting in a fast and efficient calculation of **9**, demonstrating the utility of this function across various data formats in Python.

Summary of Implementation Strategies

The choice between implementing a custom function and utilizing the SciPy library depends primarily on the context and scale of the project. For small-scale operations or environments where minimal dependencies are preferred, the custom function provides transparency and self-sufficiency. It serves as an excellent pedagogical tool for understanding the underlying mathematics, particularly the role of absolute differences.

However, for production-level code, high-performance computing, or when integrating distance metrics into complex data analysis pipelines involving Pandas DataFrames, the SciPy `cityblock()` function is unequivocally the superior choice. SciPy's functions are generally optimized using C or Fortran backends, providing significant speed improvements compared to pure Python implementations, especially when dealing with massive datasets composed of millions of data points or high-dimensional vectors.

Mastery of both techniques ensures that developers can select the most appropriate tool for any computational challenge involving the grid-based distance calculation. The Manhattan Distance remains a foundational metric, offering crucial insights in scenarios where paths are constrained or where linear feature disparity is the desired measure of separation.