Learning to Rank Data: A NumPy Array Tutorial

Authored by

Mohammed looti

October 26, 2025

Introduction to Ranking in Data Science

In the foundational processes of data analysis and statistics, assigning an ordinal rank to elements within a dataset is a critically important task. Ranking effectively sorts data, providing a relative standing for each item based on its numerical value. This technique is indispensable for numerous applications, including interpreting data distribution skewness, preparing inputs for non-parametric statistical tests, or quickly identifying outliers and high-performing records in large datasets. Understanding the position of an element relative to all others allows analysts to gain immediate insights into magnitude and importance.

When dealing with substantial volumes of numerical data in Python, efficiency is paramount. The NumPy library provides the robust, high-performance tools necessary for array manipulation and calculation. This specialized guide is dedicated to exploring two highly effective methods for calculating the rank of items stored within a NumPy array: the pure NumPy approach using the `argsort()` function, and the more statistically nuanced method provided by the `rankdata()` function, which is part of the extensive SciPy library.

While both methods achieve the objective of ranking, they differ significantly in their approach, particularly in how they manage complexity--specifically, the treatment of duplicate or **tied values**. The choice between these two powerful functions should be informed by the specific analytical requirements of your project, especially if sophisticated tie-handling methods are required. By examining practical, side-by-side examples, we aim to provide a clear understanding of the implementation nuances and the resulting rank outputs generated by each technique, ensuring you select the optimal tool for your data science workflow.

Defining the Sample Array for Demonstration

To ensure a consistent and clear demonstration of both the `argsort()` and `rankdata()` ranking procedures, we will rely on a single, fixed sample array throughout all our examples. This array is intentionally constructed to contain a mix of unique numerical values and crucial duplicate values. The presence of these duplicates is essential, as it allows us to effectively highlight and compare the distinct ways in which each function addresses tie-handling scenarios, which is often the most complex aspect of array ranking.

The following Python code snippet initializes our sample array using the NumPy library and displays its contents. Note that the array contains six elements, with the maximum value, `9`, appearing twice at the end of the sequence.

import numpy as np

Define array of values

```
my_array = np.array()

# View array
print(my_array)
```

Our objective is to transform this raw array into a new array containing the rank of each corresponding element from the original sequence. In standard ranking conventions, the smallest value in the dataset receives the lowest rank (Rank 1, or Rank 0 if using 0-based indexing), and ranks increase sequentially as the values grow larger. Observing how the two tied values (the two `9`s) are assigned their final ranks will be key to distinguishing the capabilities of the two methods discussed below.

Method 1: Ranking Using NumPy's Double `argsort()` Trick

The first, and purely NumPy-based, method for ranking array elements involves a clever application of the `argsort()` function. Unlike specialized ranking functions, `argsort()` does not directly return ranks; rather, it returns the indices that would correctly sort the array into ascending order. By executing `argsort()` twice consecutively, we can effectively obtain the desired ranks.

The mechanism functions in two distinct steps. The initial call to `argsort()` identifies the positions of the values if the array were sorted. For example, if the smallest value is found at index 3 in the original array, the first element of the result will be 3. The second application of `argsort()` on this intermediate result then uses these sorted indices to map back to the original positions, thereby translating the sorting order into a rank assignment.

Applying this efficient technique to our `my_array` yields a sequence of 0-based ranks, where 0 is assigned to the smallest element. This pure NumPy solution is highly desirable when minimizing external dependencies is a priority.

```
# Calculate rank of each item in array
ranks = np.array(my_array).argsort().argsort()

# View ranks
print(ranks)
```

The resulting ranks indicate that the element at index 3 (value 1) has rank 0, and the element at index 5 (the second 9) has the highest rank of 5. A crucial characteristic of `argsort()` when used for ranking is its fixed tie-handling behavior: it utilizes an **ordinal** method. This means that if two values are identical, the one that appears earlier in the original array is assigned a lower rank,

regardless of the value. In our example, `my_array` (9) gets rank 4, and `my_array` (9) gets rank 5 because it appeared later in the sequence. While simple and dependency-free, this method lacks the flexibility required by advanced statistical analyses that mandate different tie-breaking rules.

Method 2: Leveraging SciPy's Dedicated `rankdata()` Function

For analytical situations demanding greater control over ranking methodologies, especially concerning the statistical treatment of tied values, the `rankdata()` function, available within SciPy's `stats` module, is the superior choice. This function is specifically engineered to compute various forms of ranks and offers multiple parameters to dictate how duplicates are resolved, moving beyond the simple ordinal behavior of `argsort()`.

To employ this function, the first step is to import it from `scipy.stats`. Then, you pass the target NumPy array directly to `rankdata()` without any subsequent operations. The default behavior provides statistically sound ranks, typically starting from 1 (1-based indexing).

from scipy.stats import rankdata

Calculate rank of each item in array
ranks = rankdata(my_array)

# View ranks
print(ranks)

array()

The resulting array, `array()`, reveals two major distinctions from the `argsort()` method. Firstly, the ranks are **1-based** by default, meaning the smallest value (1) receives a rank of 1.0. Secondly, and most importantly, the tied values (the two 9s) are assigned an **average rank**. Since these two values would normally occupy the 5th and 6th positions in a sorted list, `rankdata()` assigns both the fractional rank of `(5 + 6) / 2 = 5.5`. This 'average' method is the standard for many statistical procedures.

If project specifications require consistency with 0-based indexing, similar to NumPy's conventions, you can easily adjust the output by subtracting 1 from the result of the `rankdata()` call. This simple arithmetic operation shifts the entire ranking scale while preserving the essential statistical integrity of the average rank calculation for tied values.

from scipy.stats import rankdata

Calculate rank of each item in array

```
ranks = rankdata(my_array) - 1

# View ranks
print(ranks)
```

Advanced Tie-Handling with `rankdata()` Methods

The true power and flexibility of SciPy's `rankdata()` function stem from its ability to handle tied values using various methods, controlled via the `method` argument. While 'average' is the default and mathematically sound for most statistical tests, other analytical contexts may necessitate different tie-breaking conventions. This versatility is often the deciding factor in choosing `rankdata()` over the fixed behavior of NumPy's `argsort()`.

For example, if you explicitly require the behavior exhibited by the double `argsort()` method--where the rank of tied values is determined by their sequential order of appearance--you can pass `method='ordinal'` to the function. This configuration ensures that even when using the specialized SciPy function, you can replicate the specific output generated by the base NumPy approach.

```
from scipy.stats import rankdata

# Calculate rank of each item in array using 'ordinal' method
ranks = rankdata(my_array, method='ordinal') - 1

# View ranks
print(ranks)
```

The output matches the result from our earlier double `argsort()` example, demonstrating that `rankdata()` can be customized to fit various ranking conventions. Below is a summary of the available tie-handling methods provided by `rankdata()`:

`'average'` (Default): Assigns the mean of the ranks that the tied values would normally occupy. This produces fractional ranks and is standard for correlation analysis.

`'ordinal'`: Assigns ranks sequentially based on the order of appearance in the input array. This is useful when the original order is meaningful, mimicking the behavior of `argsort()`.

`'min'`: All tied values receive the minimum rank of the block they occupy. If tied values would be ranks 5 and 6, both receive rank 5.

`'max'`: All tied values receive the maximum rank of the block they occupy. If tied values would be ranks 5 and 6, both receive rank 6.

`'dense'`: Assigns ranks as consecutive integers without leaving gaps for the tied values. For instance, if ranks 2 and 3 are tied, the next unique value receives rank 3, not rank 4.

Choosing the Optimal Ranking Method

The decision between implementing the double `argsort()` technique from NumPy and utilizing SciPy's specialized `rankdata()` function hinges primarily on your project's constraints regarding external dependencies and the required statistical precision in tie-handling. Both are highly efficient for typical data array sizes, but they serve different roles in the data pipeline.

You should opt for the pure NumPy `argsort()` approach if the following conditions apply: you must maintain a minimal dependency footprint, avoiding the addition of the larger SciPy library; the default **ordinal** tie-handling mechanism, which assigns ranks based on the order of appearance for tied values, is acceptable for your analysis; and your ranking needs are computationally simple, not requiring fractional or statistically modified ranks. This method provides a fast, self-contained solution.

Conversely, selecting SciPy's `rankdata()` is advisable when statistical accuracy and flexibility are paramount. This function is essential if your project demands specific tie-handling conventions, such as assigning 'average', 'min', 'max', or 'dense' ranks to duplicates. Furthermore, if your analysis already relies on other advanced statistical or scientific computing tools provided by the SciPy ecosystem, integrating `rankdata()` is a natural and semantically clear choice, offering direct control over rank computation parameters.

Additional Resources for NumPy and SciPy

To deepen your expertise in high-performance numerical computing and array manipulation, the following official documentation and guides offer comprehensive information on the functions discussed and the wider capabilities of the NumPy and SciPy libraries.

NumPy User Guide: Essential reading for new users and a detailed reference for foundational concepts.

NumPy `argsort()` Documentation: Official documentation detailing the parameters and exact behavior of the array sorting index function.

SciPy `rankdata()` Documentation: A comprehensive guide to all ranking methods, tie-handling options, and use cases for the function.

SciPy Reference Guide: Explore the vast collection of scientific and statistical tools available within the SciPy library beyond basic array ranking.