Aggregations and Sorting in NumPy by Mrittika Megaraj

Aggregations: Min, Max, and Everything In Between

When dealing with a significant volume of data, it is common practice to first generate some kind of summary statistics. The mean and standard deviation are two of the most often used summary statistics because they provide an overview of the "usual" values in a dataset (the sum, product, median, minimum and maximum, quantiles, etc.).

In this study, we'll go over some of the built-in aggregating methods available in NumPy and show you how quick they can make your work with arrays.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function:

```
import numpy as np
L = np.random.random(100)
sum(L)
55.61209116604941
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same in the simplest case:

```
np.sum(L)
55.612091166049424
```

However, because it executes the operation in compiled code, NumPy's version of the operation is computed much more quickly:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings, and `np.sum` is aware of multiple array dimensions, as we will see in the following section.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
min(big_array), max(big_array)
(1.1717128136634614e-06, 0.9999976784968716)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

```
np.min(big_array), np.max(big_array)
(1.1717128136634614e-06, 0.9999976784968716)
```

```
%timeit min(big_array)
%timeit np.min(big_array)
10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 µs per loop
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
1.17171281366e-06 0.999997678497 499911.628197
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

Multi dimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
M = np.random.random((3, 4))
print(M)
```

```
[[ 0.8967576   0.03783739  0.75952519  0.06682827]
 [ 0.8354065   0.99196818  0.19544769  0.43447084]
 [ 0.66859307  0.15038721  0.37911423  0.6687194 ]]
```

By default, each NumPy aggregation function will return the aggregate over the entire array:

```
M.sum()
6.0850555667307118
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

```
M.min(axis=0)
array([ 0.66859307,  0.03783739,  0.19544769,  0.06682827])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

```
M.max(axis=1)
array([ 0.8967576 ,  0.99196818,  0.6687194 ])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

Other aggregation functions

NumPy provides many other aggregation functions, but we won't discuss them in detail here. Additionally, most aggregates have a `NaN`-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value (for a fuller discussion of missing data, see Handling Missing Data). Some of these `NaN`-safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the study.

Example: What is the Average Height of US Presidents?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

```
!head -4 data/president_heights.csv
order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
```

We'll use the Pandas package, which we'll explore more fully later on , to read the file and extract this information (note that the heights are measured in centimeters).

```
import pandas as pd
data = pd.read_csv('data/president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)
```

```
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Now that we have this data array, we can compute a variety of summary statistics:

```
print("Mean height:       ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:    ", heights.min())
print("Maximum height:    ", heights.max())
```

```
Mean height:        179.738095238
Standard deviation: 6.93184344275
Minimum height:     163
Maximum height:     193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

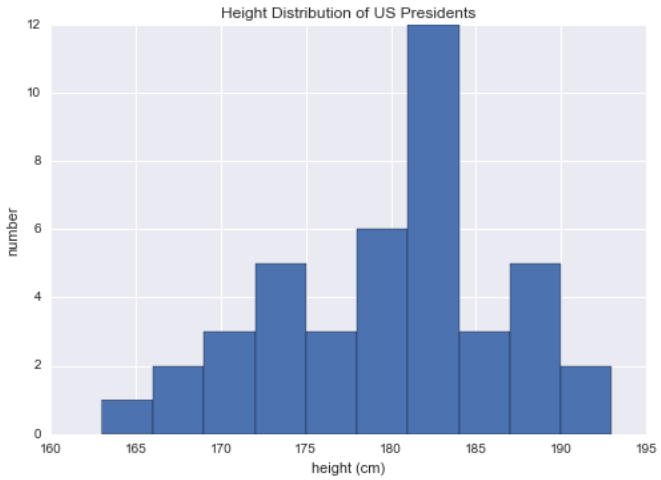
```
print("25th percentile:   ", np.percentile(heights, 25))
print("Median:            ", np.median(heights))
print("75th percentile:   ", np.percentile(heights, 75))
```

```
25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a visual representation of this data, which we can accomplish using tools in Matplotlib (we'll discuss Matplotlib more fully in later on). For example, this code generates the following chart:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()  # set plot style
plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```



These aggregates are some of the fundamental pieces of exploratory data analysis that we'll explore in more depth in later chapters of the study.

Sorting Arrays

NumPy array access and manipulation tools have been the primary focus so far. Algorithms for arranging numerical data in NumPy arrays are discussed here. If you've ever taken an introductory CS course, you've undoubtedly dreamed (or nightmared) about various sorting algorithms including insertion sorts, selection sorts, merge sorts, rapid sorts, bubble sorts, and many more. Every one of these approaches may be thought of as somewhat different ways to do the same basic operation of sorting the elements of a list or array.

A basic selection sort, for instance, iteratively searches for the list's minimal value and replaces it with another until the list is sorted. Using just a few lines of Python, we can do this:

```
import numpy as np

def selection_sort(x):
    for i in range(len(x)):
        swap = i + np.argmin(x[i:])
        (x[i], x[swap]) = (x[swap], x[i])
    return x


x = np.array([2, 1, 4, 3, 5])
selection_sort(x)
```

```
array([1, 2, 3, 4, 5])
```

The selection sort is great for small arrays due to its ease of use, but it is too slow to be practical for anything bigger, as any first-year computer science student will tell you. To get the swap value from a list of N values, you'll need to do N loops, each of which involves on order N comparisons. "big-O" notation is often used to describe these algorithms, and selection sort averages fall within this category. $O[N2]$: if you double the number of items in the list, the execution time will go up by about a factor of four.

Even selection sort, though, is much better than my all-time favorite sorting algorithms, the *bogosort*:

```
def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x


x = np.array([2, 1, 4, 3, 5])
bogosort(x)


array([1, 2, 3, 4, 5])
```

This silly sorting method relies on pure chance: it repeatedly applies a random shuffling of the array until the result happens to be sorted. With an average scaling of $\mathcal{O}[N \times N!]$, (that's *N* times *N* factorial) this should–quite obviously–never be used for any real computation.

Fortunately, Python contains built-in sorting algorithms that are *much* more efficient than either of the simplistic algorithms just shown. We'll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

Fast Sorting in NumPy: np.sort and np.argsort

Although Python has built-in sort and sorted functions to work with lists, we won't discuss them here because NumPy's np.sort function turns out to be much more efficient and useful for our purposes. By default np.sort uses an $\mathcal{O}[N \log N]$, *quicksort* algorithm, though *mergesort* and *heapsort* are also available. For most applications, the default quicksort is more than sufficient.to return a sorted version of the array without modifying the input, you can use np.sort:

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)


array([1, 2, 3, 4, 5])
```

If you prefer to sort the array in-place, you can instead use the sort method of arrays:

```
x.sort()
print(x)
```

```
[1 2 3 4 5]
```

A related function is argsort, which instead returns the *indices* of the sorted elements:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

```
[1 0 3 2 4]
```

This result's first value represents the index of the smallest element, the second value represents the index of the next-smallest, and so on. The sorted array may then be built using these indices.

```
x[i]
```

```
array([1, 2, 3, 4, 5])
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
```

```
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]
```

```
# sort each column of X
np.sort(X, axis=0)
```

```
array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])
```

```
# sort each row of X
np.sort(X, axis=1)
```

```
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])
```

Remember that this method treats each row or column as a separate array, breaking any dependencies between the values.

Partial Sorts: Partitioning

Sometimes, rather of sorting the whole array, we just care about the k smallest values. This is something that may be done using the np.partition function in NumPy. For an input array and a parameter K, np.partition returns a new array with the smallest K values on the left and the remaining values, in any order, on the right.

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
```

```
array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

```
np.partition(X, 2, axis=1)
```

```
array([[3, 4, 6, 7, 6, 9],
       [2, 3, 4, 7, 6, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 9, 5]])
```

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a np.argsort that computes indices of the sort, there is a np.argpartition that computes indices of the partition. We'll see this in action in the following section.