Numpy:Case Study

Lists Of Lists for CSV Data

- Before using NumPy, we'll first try to work with the data using Python and the csv package.

- We can read in the file using the csv.reader object, which will allow us to read in and split up all the content from the *ssv* file.

Reading csv file

- import numpy as np
- wines=np.genfromtxt("winequality-red.csv", delimiter=";",skip_header=1)
- wines

Example Without Numpy Arrays

- To print first three rows :

```
print(wines[:3])
```

Problem:

- Extract the last element from each row after the header row
- Convert each extracted element to a float
- Assign all the extracted elements to the list qualities
- Divide the sum of all the elements in qualities by the total number of elements in qualities to get the mean

Solution:

```
qualities = [float(item[-1]) for item in wines[1:]]
sum(qualities) / len(qualities)
```

Numpy 2-Dimensional Arrays

- Although we were able to do the calculation we wanted, the code is fairly complex, and it won't be fun to have to do something similar every time we want to compute a quantity. Luckily, we can use NumPy to make it easier to work with our data.
- A 2-dimensional array is also known as a matrix, and is something you should be familiar with. In fact, it's just a different way of thinking about a list of lists. A matrix has rows and columns. By specifying a row number and a column number, we're able to extract an element from a matrix.
- If we picked the element at the first row and the second column, we'd get volatile acidity. If we picked the element in the third row and the second column, we'd get 0.88

Creating A NumPy Array

- In a NumPy array, the number of dimensions is called the rank, and each dimension is called an axis. So the rows are the first axis, and the columns are the second axis.
- If we pass in a list of lists, it will automatically create a NumPy array with the same number of rows and columns.
- Because we want all of the elements in the array to be *float* elements for easy computation, we'll leave off the header row, which contains *strings*.
- One of the limitations of NumPy is that all the elements in an array have to be of the same type, so if we include the header row, all the elements in the array will be read in as strings. Because we want to be able to do computations like find the average quality of the wines, we need the elements to all be floats.

Creating A NumPy Array

- Pass the list of lists wines into the array function, which converts it into a NumPy array.Exclude the header row with list slicing.
- Specify the keyword argument dtype to make sure each element is converted to a float.
- import numpy as np
- wines = np.array(wines[1:], dtype=np.float)
- Wines
- Shape(wines)
- 1599,12

Alternative NumPy Array Creation Methods

- It‘s possible to use NumPy to directly read csv or other files into arrays. We can do this using the np.genfromtxt function.
- Use the genfromtxt function to read in the winequality-red.csv file.
- Specify the keyword argument delimiter=";" so that the fields are parsed properly.
- Specify the keyword argument skip_header=1 so that the header row is skipped.
- wines = np.genfromtxt("winequality-red.csv", delimiter=";", skip_header=1)
- wines will end up looking the same as if we read it into a list then converted it to an array of floats. NumPy will automatically pick a data type for the elements in an array based on their format.
-

Indexing NumPy Arrays

- Let's select the element at row 3 and column 4. In the below code, we pass in the index 2 as the row index, and the index 3 as the column index. This retrieves the value from the fourth column of the third row:
- wines[2,3]
- 2.2999999999999998
- Since we're working with a 2-dimensional array in NumPy, we specify 2 indexes to retrieve an element. The first index is the row, or axis 1, index, and the second index is the column, or axis 2, index. Any element in wines can be retrieved using 2 indexes.

Slicing NumPy Arrays

- If we want to select the first three items from the fourth column, we can do it using a colon (:).
- A colon indicates that we want to select all the elements from the starting index up to but not including the ending index.
- This is also known as a slice:
- wines[0:3,3]
- array([1.9, 2.6, 2.3])

Slicing NumPy Arrays

- Just like with list slicing, it's possible to omit the 0 to just retrieve all the elements from the beginning up to element 3:
- wines[:3,3]
- array([1.9, 2.6, 2.3])
-
- With no starting or ending indices. The below code will select the entire fourth column:
- wines[:,3]
- array([1.9, 2.6, 2.3, ..., 2.3, 2. , 3.6])

Slicing NumPy Arrays

- We selected an entire column above, but we can also extract an entire row:
- wines[3,:]
- array([11.2 ,   0.28 ,   0.56 ,   1.9 ,   0.075, 17.   , 60.   ,
-         0.998,   3.16 ,   0.58 ,   9.8 ,   6.   ])
- If we take our indexing to the extreme, we can select the entire array using two colons to select all the rows and columns in wines. This is a great party trick, but doesn't have a lot of good applications:
- wines[:,:]
-

Assigning Values To NumPy Arrays

- We can also use indexing to assign values to certain elements in arrays. We can do this by assigning directly to the indexed value:
- wines[1,5] = 10
- We can do the same for slices. To overwrite an entire column, we can do this:
- wines[:,10] = 50
- The above code overwrites all the values in the eleventh column with 50.

1-Dimensional NumPy Arrays

- NumPy is a package for working with multidimensional arrays.
- One of the most common types of multidimensional arrays is the 1-dimensional array, or vector.
- When we sliced wines, we retrieved a 1-dimensional array.
- A 1-dimensional array only needs a single index to retrieve an element.
- Each row and column in a 2-dimensional array is a 1-dimensional array.
- Just like a list of lists is analogous to a 2-dimensional array, a single list is analogous to a 1-dimensional array. If we slice wines and only retrieve the third row, we get a 1-dimensional array:
- third_wine = wines[3,:]

1-Dimensional NumPy Arrays

- Here's how third_wine looks:
- 11.200
- 0.280
- 0.560
- 1.900
- 0.075
- 17.000
- 60.000
- 0.998
- 3.160
- 0.580
- 9.800
- 6.000

1-Dimensional NumPy Arrays

- We can retrieve individual elements from third_wine using a single index. The below code will display the second item in third_wine:
- third_wine[1]
- 0.28000000000000003
-

Converting datatypes

- As you can see above, all of the items in the resulting array are integers. Note that we used the Python int type instead of a NumPy data type when converting wines. This is because several Python data types, including float, int, and string, can be used with NumPy, and are automatically converted to NumPy data types.
- We can check the name property of the dtype of the resulting array to see what data type NumPy mapped the resulting array to:
- int_wines = wines.astype(int)
- int_wines.dtype.name
- Returns ‘int64'

Converting datatypes

- The array has been converted to a 64-bit integer data type. This allows for very long integer values, but takes up more space in memory than storing the values as 32-bit integers.
- If you want more control over how the array is stored in memory, you can directly create NumPy dtype objects like numpy.int32:
  - np.int32
- You can use these directly to convert between types:

```
wines.astype(np.int32)
```

Single Array Math

- If you do any of the basic mathematical operations (/, *, -, +, ^) with an array and a value, it will apply the operation to each of the elements in the array.
- Let's say we want to add 10 points to each quality score Here's how we'd do that:
- wines[:,11] + 10

Shows array([15., 15., 15., ..., 16., 15., 16.])

- Note that the above operation won't change the wines array -- it will return a new 1-dimensional array where 10 has been added to each element in the quality column of wines.
- If we instead did +=, we'd modify the array in place:
- wines[:,11] += 10 wines[:,11]
- array([15., 15., 15., ..., 16., 15., 16.])

Single Array Math

- All the other operations work the same way. For example, if we want to multiply each of the quality score by 2, we could do it like this:
- wines[:,11] * 2
- array([30., 30., 30., ..., 32., 30., 32.])

Multiple Array Math

- It's also possible to do mathematical operations between arrays. This will apply the operation to pairs of elements. For example, if we add the quality column to itself, here's what we get:
- a=np.array(wines[:,11] + wines[:,11])
- array([10., 10., 10., ..., 12., 10., 12.])
- a.max()
- Returns the max value
- Note that this is equivalent to wines[11] * 2 -- this is because NumPy adds each pair of elements. The first element in the first array is added to the first element in the second array, the second to the second, and so on.

Multiple Array Math

- We can also use this to multiply arrays. Let's say we want to pick a wine that maximizes alcohol content and quality.
- We'd multiply alcohol by quality, and select the wine with the highest score:
- wines[:,10] * wines[:,11]
- array([47., 49., 49., ..., 66., 51., 66.])
- All of the common operations (/, *, -, +, ^) will work between arrays.

Broadcasting

- Unless the arrays that you're operating on are the exact same size, it's not possible to do element wise operations. In cases like this, NumPy performs broadcasting to try to match up elements. Essentially, broadcasting involves a few steps:
- The last dimension of each array is compared.
  - If the dimension lengths are equal, or one of the dimensions is of length 1, then we keep going.
  - If the dimension lengths aren't equal, and none of the dimensions have length 1, then there's an error.
- Continue checking dimensions until the shortest array is out of dimensions.

Broadcasting

- for example, the following two shapes are compatible:

- A: (50,3) B (3,)
- This is because the length of the trailing dimension of array A is 3, and the length of the trailing dimension of array B is 3. They're equal, so that dimension is okay. Array B is then out of elements, so we're okay, and the arrays are compatible for mathematical operations.
- The following two shapes are also compatible:
- A: (1,2) B (50,2)
- As the last dimension matches, and A is of length 1 in the first dimension.

Broadcasting

- These two arrays don't match:

  A: (50,50) B: (49,49)
- The lengths of the dimensions aren't equal, and neither array has either dimension length equal to 1.
- wines * np.array([1,2]) will produce a value error
- The above example didn't work because the two arrays don't have a matching trailing dimension. Here's an example where the last dimension does match:
- array_one = np.array([[1,2], [3,4]])
- array_two = np.array([4,5])
- array_one + array_two
- Shows array([[5, 7], [7, 9]])

Broadcasting

- As you can see, array_two has been broadcasted across each row of array_one. Here's an example with our wines data:
- rand_array = np.random.rand(12)
- wines + rand_array
- Elements of rand_array are broadcast over each row of wines, so the first column of wines has the first value in rand_array added to it, and so on.

NumPy Array Methods

- In addition to the common mathematical operations, NumPy also has several methods that you can use for more complex calculations on arrays. An example of this is the numpy.ndarray.sum method. This finds the sum of all the elements in an array by default:
- wines[:,11].sum()
- 9012.0
- wines.sum(axis=0)
- array([13303.1 , 843.985 , 433.29 , 4059.55 , 139.859 , 25384. , 74302. , 1593.79794, 5294.47 , 1052.38 , 16666.35 , 9012.])
- We can verify that we did the sum correctly by checking the shape. The shape should be 12, corresponding to the number of columns:
- wines.sum(axis=0).shape
- (12,)

NumPy Array Methods

- If we pass in axis=1, we'll find the sums over the second axis of the array. This will give us the sum of each row:
- wines.sum(axis=1)
- array([74.5438 , 123.0548 , 99.699 , ..., 100.48174, 105.21547, 92.49249])

NumPy Array Methods

- There are several other methods that behave like the sum method, including:
- numpy.ndarray.mean — finds the mean of an array.
- numpy.ndarray.std — finds the standard deviation of an array.
- numpy.ndarray.min — finds the minimum value in an array.
- numpy.ndarray.max — finds the maximum value in an array.

NumPy Array Comparisons

- NumPy makes it possible to test to see if rows match certain values using mathematical comparison operations like <, >, >=, <=, and ==.
- For example, if we want to see which wines have a quality rating higher than 5, we can do this:
- wines[:,11] > 5
- array([False, False, False, ..., True, False, True], dtype=bool)
- We get a Boolean array that tells us which of the wines have a quality rating greater than 5.
- We can do something similar with the other operators. For instance, we can see if any wines have a quality rating equal to 10:
- wines[:,11] == 10
- array([False, False, False, ..., False, False, False], dtype=bool)

Subsetting

- One of the powerful things we can do with a Boolean array and a NumPy array is select only certain rows or columns in the NumPy array.
- For example, the below code will only select rows in wines where the quality is over 7:
  - high_quality = wines[:,11] > 7
  - wines[high_quality,:][:3,:]

Subsetting

- We select only the rows where high_quality contains a True value, and all of the columns.
- This subsetting makes it simple to filter arrays for certain criteria. For example, we can look for wines with a lot of alcohol and high quality.
- In order to specify multiple conditions, we have to place each condition in parentheses, and separate conditions with an ampersand (&):
- high_quality_and_alcohol = (wines[:,10] > 10) & (wines[:,11] > 7) wines[high_quality_and_alcohol,10:]
- array([[12.8, 8.], [12.6, 8.], [12.9, 8.], [13.4, 8.], [11.7, 8.], [11. , 8.], [11. , 8.], [14. , 8.], [12.7, 8.], [12.5, 8.], [11.8, 8.], [13.1, 8.], [11.7, 8.], [14. , 8.], [11.3, 8.], [11.4, 8.]])

Subsetting

- We can combine subsetting and assignment to overwrite certain values in an array:

-
```
high_quality_and_alcohol = (wines[:,10] > 10) & (wines[:,11] > 7)
wines[high_quality_and_alcohol,10:] = 20
```

Reshaping NumPy Arrays

- We can change the shape of arrays while still preserving all of their elements.
- This often can make it easier to access array elements. The simplest reshaping is to flip the axes, so rows become columns, and vice versa.
- We can accomplish this with the numpy.transpose function:

- np.transpose(wines).shape
- (12,1599)

Numpy.ravel function

- It turns an array into a one-dimensional representation. It will essentially flatten an array into a long sequence of values:
- wines.ravel()
- array_one = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
- array_one.ravel()
- Array([1,,3,4,5,6,7,8])
- Finally, we can use the numpy.reshape function to reshape an array to a certain shape we specify. The below code will turn the second row of wines into a 2-dimensional array with 2 rows and 6 columns.
- Wines[1,:].Reshape((2,6))

Combining NumPy Arrays with vstack

- With NumPy, it's very common to combine multiple arrays into a single unified array. It can be achieved by using Numpy.vstack to vertically stack multiple arrays.
- Think of it like the second arrays's items being added as new rows to the first array. We can read in the winquality-wine.csv dataset that contains information on the quality of white wines, then combine it with our existing dataset,wines, which contains information on red wines.
- The arrays must have the same shape along all but the first axis.
- white_wines = np.genfromtxt("winequality-white.csv", delimiter=";", skip_header=1) white_wines.shape
- (4898,12)

Combining NumPy Arrays with vstack

- we have the white wines data, we can combine all the wine data.
- Use the vstack function to combine wines and white_wines .
- Display the shape of the result
- all_wines = np.vstack((wines, white_wines))
- all_wines.shape
- (6497,12)

Combining NumPy Arrays with hstack

- If we want to combine arrays horizontally, where the number of rows stay constant, but the columns are joined, then we can use the Numpy.hstack function.
- The arrays we combine need to have the same number of rows for this to work.
- Finally, we can use numpy.concatenate as a general purpose version of hstack and vstack.
- If we want to concatenate two arrays, we pass them into concatenate then specify the axis keyword argument that we want to concatenate along. Concatenating along the first axis is similar to vstack and concatenating along the second axis is similar to hstack
- np.concatenate((wines, white_wines), axis=0)

Exercise

- Create a 3 x 4 array filled with all zeros, and a 6 x 4 array filled with all 1s.
- Concatenate both arrays vertically into a 9 x 4 array, with the all zeros array on top.
- Assign the entire first column of the combined array to first_column.
- Print the first_column.

Reference

- If you want to dive into more depth, here are some resources that may be helpful:
- NumPy Quickstart -- has good code examples and covers most basic NumPy functionality.
- Python NumPy Tutorial -- a great tutorial on NumPy and other Python libraries.
- Visual NumPy Introduction -- a guide that uses the game of life to illustrate NumPy concepts.