Case Study - Cricket Tournament

Example - 1 Players list contain the height(inches) and weight(lbs) data for all the players

1- Create an random interger 2D np array of shape 1015*2 representiing height and weight of players. heights in inches should be in the range of 67 to 83 wights in lbs should be in the range of 150 to 290

hint: vstack

```
import numpy as np
import pprint
```

```
# Define list
heights = [74, 74, 72, 72, 73, 69, 69, 71, 76, 71, 73, 73, 74, 74, 69, 70, 73, 75, 78, 79, 76, 74, 76, 72, 71, 75, 77, 74, 73, 74, 78, 73, 75
```

```
weights = [180, 215, 210, 210, 188, 176, 209, 200, 231, 180, 188, 180, 185, 160, 180, 185, 189, 185, 219, 230, 205, 230, 195, 180, 192, 225,
```

```
weights_lbs = np.array(weights)
heights_inches = np.array(heights)
```

Convert the heights to meters and weights to kg

```
height_meters = heights_inches * 0.0254
weights_kgs = weights_lbs * 0.45359237
```

```
pprint.pprint({
    'height_meters': height_meters,
    'weights_kgs': weights_kgs
})
print()
pprint.pprint({
    'weights_lbs': weights_lbs,
    'heights_inches': heights_inches
})
```

```
 {'height_meters': array([1.8796, 1.8796, 1.8288, ..., 1.905 , 1.905 , 1.8542]),
  'weights_kgs': array([81.6466266 , 97.52235955, 95.2543977 , ..., 92.98643585,
        86.1825503 , 88.45051215])}

 {'heights_inches': array([74, 74, 72, ..., 75, 75, 73]),
  'weights_lbs': array([180, 215, 210, ..., 205, 190, 195])}
```

```
height_meters.size
```

```
1015
```

```
weights_kgs.size
```

```
1015
```

```
players = np.column_stack((height_meters, weights_kgs))
players
# players = np.hstack((height_meters.reshape((1015,1)), weights_kgs.reshape(1015,1)))
# players.shape
```

```
array([[ 1.8796    , 81.6466266 ],
       [ 1.8796    , 97.52235955],
       [ 1.8288    , 95.2543977 ],
       ...,
       [ 1.905     , 92.98643585],
       [ 1.905     , 86.1825503 ],
       [ 1.8542    , 88.45051215]])
```

```
players.shape
```

```
(1015, 2)
```

Fetch the first row from the array

```
players[0]
```

```
array([ 1.8796   , 81.6466266])
```

Fetch the first row 2nd element from the array

```
players[0, 1]
```

```
81.6466266
```

Fetch the first column from the array

```
players[:, 0]
```

```
array([1.8796, 1.8796, 1.8288, ..., 1.905 , 1.905 , 1.8542])
```

Fetch the height (1st column) of 125th player from the array

```
players[125, 0]
```

```
1.8541999999999998
```

Fetch height and weight of players with height above 1.8m

```
# Method 1
filtered_players = players[players[:, 0] > 1.8]
print(filtered_players)
print(filtered_players.shape)
```

```
[[ 1.8796     81.6466266 ]
 [ 1.8796     97.52235955]
 [ 1.8288     95.2543977 ]
 ...
 [ 1.905      92.98643585]
 [ 1.905      86.1825503 ]
 [ 1.8542     88.45051215]]
(936, 2)
```

```
# Method 2
filtered_players2 = players[height_meters > 1.8]
print(filtered_players2)
print(filtered_players2.shape)
```

```
[[ 1.8796     81.6466266 ]
 [ 1.8796     97.52235955]
 [ 1.8288     95.2543977 ]
 ...
 [ 1.905      92.98643585]
 [ 1.905      86.1825503 ]
 [ 1.8542     88.45051215]]
(936, 2)
```

```
# Both method yield same output
np.array_equal(filtered_players, filtered_players2)
```

```
True
```

Skills Array of size 1015 - holds the player key skills with given skills ['Batsman', 'Bowler', 'Keeper', 'Keeper-Batsman']

hint: use np.tile

```
skillset = ['Batsman', 'Bowler', 'Keeper', 'Keeper-Batsman']
skills = np.tile(skillset, 254)[:1015]
print(skills.shape)
```

```
(1015,)
```

Fetch Heights of the Batsmen

```
batters = players[skills == 'Batsman']
# print(batters)
print("Batsmen shape:", batters.shape)
print()

batters_heights = batters[:, 0]
# print(batters_heights)
print(f"Batsmen heights (Shape: {batters_heights.shape}): \n", batters_heights)
```

```
Batsmen shape: (254, 2)

Batsmen heights (Shape: (254,)):
 [1.8796 1.8542 1.9304 1.8796 1.8542 1.9304 1.8034 1.8542 1.905  1.8796
 1.8542 1.8796 1.8796 1.9304 1.8796 1.9812 1.9304 1.905  1.778  1.7272
 1.905  1.8796 1.9304 1.905  1.8796 1.8542 1.7526 1.905  1.9304 1.8796
 2.032  1.9812 1.905  1.8796 1.8542 1.8542 1.8034 1.8796 1.905  1.905
 1.9812 1.8796 1.9812 1.8288 1.778  1.8542 1.8288 1.905  1.9304 1.905
 1.905  1.8542 1.8796 1.905  1.8796 1.9558 1.9812 1.905  1.9304 1.7526
 1.8288 1.8034 1.8288 1.8288 1.9558 1.8034 1.905  1.8796 1.778  1.8796
 1.8034 1.8034 1.905  1.9304 1.8288 1.8034 1.9812 1.8796 1.905  1.8542
 1.8288 1.8796 1.8796 1.8034 1.8542 1.9558 1.8796 1.778  1.8542 2.032
 1.8796 1.905  1.8796 1.8796 1.905  1.8542 1.8288 1.8796 1.8796 1.8796
 1.905  1.8796 1.8796 1.8034 1.778  1.9304 1.905  2.0066 1.905  1.778
 1.8542 1.8288 1.7526 1.8288 1.9304 1.9304 1.8034 1.905  1.8288 1.8542
 1.8542 1.905  1.905  2.032  1.8796 1.8542 1.9558 1.8542 1.8288 1.778
 1.8542 1.8796 1.905  1.8542 1.8542 1.905  1.8796 1.8288 1.8288 1.8796
 1.8796 1.8796 1.8542 1.8796 1.8796 1.8034 1.905  2.0066 1.9812 1.8288
 1.8034 1.8796 1.8034 1.8288 1.8034 1.778  2.0066 1.905  1.8542 1.778
 1.8796 1.8288 1.8288 1.8796 1.8288 2.032  1.8034 1.9558 1.905  1.7526
 1.8542 1.9304 2.0066 1.8796 1.9304 1.8288 1.8796 1.9304 1.8542 1.8034
 1.8796 1.8288 1.8288 1.8796 1.8542 1.8288 1.8034 1.8542 1.905  1.7272
 1.9558 2.0574 1.8288 1.8542 1.8034 1.9304 1.7272 1.9558 1.905  1.8288
 1.8288 1.778  1.8034 1.8034 1.8288 1.8288 1.8542 1.905  1.905  1.8542
 1.905  1.9304 1.8796 1.8034 1.8288 1.9304 1.8542 1.9558 1.8542 1.8542
 1.9304 1.8288 1.8542 1.905  1.9558 1.905  1.8034 1.9304 1.9304 1.905
 1.8288 1.8796 1.8034 1.8796 1.905  1.9558 1.9558 1.8288 1.8542 1.8542
 1.778  1.8796 1.9558 1.8796 1.9812 1.8796 1.8034 1.7018 1.8542 1.8288
 1.8796 1.9304 1.8288 1.905 ]
```

✓ 0s completed at 10:05 AM

NumPy Practice

This notebook offers a set of exercises for different tasks with NumPy.

It should be noted there may be more than one different way to answer a question or complete an exercise.

Exercises are based off (and directly taken from) the quick introduction to NumPy notebook.

Different tasks will be detailed by comments or text.

For further reference and resources, it's advised to check out the NumPy documentation.

And if you get stuck, try searching for a question in the following format: "how to do XYZ with numpy", where XYZ is the function you want to leverage from NumPy.

```
# Import NumPy as its abbreviation 'np'
import numpy as np
import pprint
```

```
# Create a 1-dimensional NumPy array using np.array()
onedarray = np.array([5,2,5])
print("1d shape: ", onedarray.shape)
print("1d: ", onedarray)
print()

# Create a 2-dimensional NumPy array using np.array()
twodarray = np.array([[5,2,5,9], [8,7,2,9], [7,9,9,2]])
print("2d shape: ", twodarray.shape)
print("2d: ", twodarray)
print()

# Create a 3-dimensional Numpy array using np.array()
threedarray = np.array([ [[5,2,5], [8,7,2]], [[9,6,5], [8,2,1]] ])
print("3d: ", threedarray)
print()
```

```
1d shape:  (3,)
1d:  [5 2 5]

2d shape:  (3, 4)
2d:  [[5 2 5 9]
 [8 7 2 9]
 [7 9 9 2]]

3d:  [[[5 2 5]
  [8 7 2]]

 [[9 6 5]
  [8 2 1]]]
```

Now we've you've created 3 different arrays, let's find details about them.

Find the shape, number of dimensions, data type, size and type of each array.

```
# Attributes of 1-dimensional array (shape,
# number of dimensions, data type, size and type)
print("1d shape: ", onedarray.shape)
print("1d ndim: ", onedarray.ndim)
print("1d dtype: ", onedarray.dtype)
print("1d size: ", onedarray.size)
print("1d type: ", type(onedarray))
```

```
1d shape:  (3,)
1d ndim:  1
1d dtype:  int64
1d size:  3
1d type:  <class 'numpy.ndarray'>
```

```
# Attributes of 2-dimensional array
print("2d shape: ", twodarray.shape)
print("2d ndim: ", twodarray.ndim)
```

```
print("2d dtype: ", twodarray.dtype)
print("2d size: ", twodarray.size)
print("2d type: ", type(twodarray))
```

```
2d shape:  (3, 4)
2d ndim:  2
2d dtype:  int64
2d size:  12
2d type:  <class 'numpy.ndarray'>
```

```
# Attributes of 3-dimensional array
print("3d shape: ", threedarray.shape)
print("3d ndim: ", threedarray.ndim)
print("3d dtype: ", threedarray.dtype)
print("3d size: ", threedarray.size)
print("3d type: ", type(threedarray))
```

```
3d shape:  (2, 2, 3)
3d ndim:  3
3d dtype:  int64
3d size:  12
3d type:  <class 'numpy.ndarray'>
```

```
# Import pandas and create a DataFrame out of one
# of the arrays you've created
import pandas as pd
df = pd.DataFrame(twodarray)
df
```

	0	1	2	3
0	5	2	5	9
1	8	7	2	9
2	7	9	9	2

```
# Create an array of shape (10, 2) with only ones
onesarr = np.ones((10,2))
print(onesarr.shape)
print(onesarr)
```

```
(10, 2)
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
```

```
# Create an array of shape (7, 2, 3) of only zeros
zerosarr = np.zeros((7,2,3))
print(zerosarr.shape)
print(zerosarr)
```

```
(7, 2, 3)
[[[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]
```

```
       [[0. 0. 0.]
        [0. 0. 0.]]]
```

```
# Create an array within a range of 0 and 100 with step 3
np.arange(0,100,3)
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48,
       51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99])
```

```
# Create a random array with numbers between 0 and 10 of size (7, 2)
np.random.randint(0,10,(7,2))
```

```
array([[9, 5],
       [6, 2],
       [8, 7],
       [1, 0],
       [9, 0],
       [4, 7],
       [0, 0]])
```

```
# Create a random array of floats between 0 & 1 of shape (3, 5)
np.random.random((3,5))
```

```
array([[0.95525542, 0.61183249, 0.48141836, 0.08770777, 0.47741059],
       [0.05117736, 0.80451785, 0.37199961, 0.26650094, 0.19018777],
       [0.78809095, 0.6841324 , 0.14258372, 0.68221247, 0.47309542]])
```

```
# Set the random seed to 42
np.random.seed(42)

# Create a random array of numbers between 0 & 10 of size (4, 6)
np.random.randint(0,10,(4,6))
```

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

Run the cell above again, what happens?

Are the numbers in the array different or the same? Why do think this is?

```
# Create an array of random numbers between 1 & 10 of size (3, 7)
# and save it to a variable
test_nums = np.random.randint(1,10,(3,7))
print(test_nums)

# Find the unique numbers in the array you just created
np.unique(test_nums)
```

```
[[7 8 3 1 4 2 8]
 [4 2 6 6 4 6 2]
 [2 4 8 7 9 8 5]]
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# Find the 0'th index of the latest array you created
test_nums[0]
```

```
array([7, 8, 3, 1, 4, 2, 8])
```

```
# Get the first 2 rows of latest array you created
test_nums[:2]
```

```
array([[7, 8, 3, 1, 4, 2, 8],
       [4, 2, 6, 6, 4, 6, 2]])
```

```
# Get the first 2 values of the first 2 rows of the latest array
test_nums[:2, :2]
```

```
array([[7, 8],
       [4, 2]])
```

```
# Create a random array of numbers between 0 & 10 and an array of ones
# both of size (3, 5), save them both to variables
random_nums = np.random.randint(0,10, (3,5))
myones = np.ones((3,5))
print(random_nums)
print(myones)
```

```
[[1 4 7 9 8]
 [8 0 8 6 8]
 [7 0 7 7 2]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

```
# Add the two arrays together
added1 = random_nums + myones
added1
```

```
array([[ 2.,  5.,  8., 10.,  9.],
       [ 9.,  1.,  9.,  7.,  9.],
       [ 8.,  1.,  8.,  8.,  3.]])
```

```
# Create another array of ones of shape (5, 3)
newones = np.ones((5,3))
newones
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
# Try add the array of ones and the other most recent array together
added1 + newones
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-42-558d4b1bea34> in <cell line: 2>()
      1 # Try add the array of ones and the other most recent array together
----> 2 added1 + newones

ValueError: operands could not be broadcast together with shapes (3,5) (5,3)
```

SEARCH STACK OVERFLOW

When you try the last cell, it produces an error. Why do think this is?

How would you fix it?

```
# Create another array of ones of shape (3, 5)
customones = np.ones((3,5))
customones
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
# Subtract the new array of ones from the other most recent array
subbedarray = added1 - customones
subbedarray
```

```
array([[1., 4., 7., 9., 8.],
       [8., 0., 8., 6., 8.],
       [7., 0., 7., 7., 2.]])
```

```
# Multiply the ones array with the latest array
multiplied = customones * subbedarray
multiplied
```

```
array([[1., 4., 7., 9., 8.],
       [8., 0., 8., 6., 8.],
       [7., 0., 7., 7., 2.]])
```

```
# Take the latest array to the power of 2 using '**'
multiplied ** 2
```

```
array([[ 1., 16., 49., 81., 64.],
       [64.,  0., 64., 36., 64.],
       [49.,  0., 49., 49.,  4.]])
```

```
# Do the same thing with np.square()
squared = np.square(multiplied)
squared
```

```
array([[ 1., 16., 49., 81., 64.],
       [64.,  0., 64., 36., 64.],
       [49.,  0., 49., 49.,  4.]])
```

```
# Find the mean of the latest array using np.mean()
np.mean(squared)
```

```
39.333333333333336
```

```
# Find the maximum of the latest array using np.max()
np.max(squared)
```

```
81.0
```

```
# Find the minimum of the latest array using np.min()
np.min(squared)
```

```
0.0
```

```
# Find the standard deviation of the latest array
np.std(squared)
```

```
26.97076606665488
```

```
# Find the variance of the latest array
np.var(squared)
```

```
727.4222222222223
```

```
# Reshape the latest array to (3, 5, 1)
reshaped = squared.reshape((3,5,1))
print(reshaped)
reshaped.shape
```

```
[[[ 1.]
  [16.]
  [49.]
  [81.]
  [64.]]

 [[64.]
  [ 0.]
  [64.]
  [36.]
  [64.]]

 [[49.]
  [ 0.]
  [49.]
  [49.]
  [ 4.]]]
(3, 5, 1)
```

```
# Transpose the latest array
transposed = np.transpose(reshaped)
print(transposed)
transposed.shape
```

```
[[[ 1. 64. 49.]
  [16.  0.  0.]
  [49. 64. 49.]
  [81. 36. 49.]
  [64. 64.  4.]]]
(1, 5, 3)
```

What does the transpose do?

```
# Create two arrays of random integers between 0 to 10
# one of size (3, 3) the other of size (3, 2)
a1 = np.random.randint(0,10, (3,3))
a2 = np.random.randint(0,10, (3,2))
print(a1)
print(a2)
```

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

```
# Perform a dot product on the two newest arrays you created
np.dot(a1,a2)
```

```
array([[52, 87],
       [40, 79],
       [14, 20]])
```

```
# Create two arrays of random integers between 0 to 10
# both of size (4, 3)
a3 = np.random.randint(0,10, (4,3))
a4 = np.random.randint(0,10, (4,3))
print(a3)
print(a4)
```

```
[[6 8 6]
 [0 0 8]
 [8 3 8]
 [2 6 5]]
[[7 8 4]
 [0 2 9]
 [7 5 7]
 [8 3 0]]
```

```
# Perform a dot product on the two newest arrays you created
np.dot(a3,a4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-84-360688785031> in <cell line: 2>()
      1 # Perform a dot product on the two newest arrays you created
----> 2 np.dot(a3,a4)

/usr/local/lib/python3.10/dist-packages/numpy/core/overrides.py in dot(*args,
**kwargs)

ValueError: shapes (4,3) and (4,3) not aligned: 3 (dim 1) != 4 (dim 0)
```

SEARCH STACK OVERFLOW

It doesn't work. How would you fix it?

```
# Take the latest two arrays, perform a transpose on one of them and then perform
# a dot product on them both
a4transposed = np.transpose(a4)
np.dot(a3,a4transposed)
```

```
array([[130,  70, 124,  72],
       [ 32,  72,  56,   0],
       [112,  78, 127,  73],
       [ 82,  57,  79,  34]])
```

Notice how performing a transpose allows the dot product to happen.

Why is this?

Checking out the documentation on `np.dot()` may help, as well as reading Math is Fun's guide on the dot product.

Let's now compare arrays.

```
# Create two arrays of random integers between 0 & 10 of the same shape
# and save them to variables
a5 = np.random.randint(0,10, (4,3))
a6 = np.random.randint(0,10, (4,3))
print(a5)
print(a6)
```

```
[[3 8 0]
 [7 6 1]
 [7 0 8]
 [8 1 6]]
[[9 2 6]
 [9 8 3]
 [0 1 0]
 [4 4 6]]
```

```
# Compare the two arrays with '>'
a5 > a6
```

```
array([[False,  True, False],
       [False, False, False],
       [ True, False,  True],
       [ True, False, False]])
```

What happens when you compare the arrays with `>`?

```
# Compare the two arrays with '>='
a5 >= a6
```

```
array([[False,  True, False],
       [False, False, False],
       [ True, False,  True],
       [ True, False,  True]])
```

```
# Find which elements of the first array are greater than 7
a5[a5 > 7]
```

```
array([8, 8, 8])
```

```
# Which parts of each array are equal? (try using '==')
a5 == a6
```

```
array([[False, False, False],
       [False, False, False],
       [False, False, False],
       [False, False,  True]])
```

```
# Sort one of the arrays you just created in ascending order
a5 = np.sort(a5)
a5
```

```
array([[0, 3, 8],
       [1, 6, 7],
       [0, 7, 8],
       [1, 6, 8]])
```

```
# Sort the indexes of one of the arrays you just created
np.argsort(a5)
```

```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

```
# Find the index with the maximum value in one of the arrays you've created
np.argmax(a5)
```

```
2
```

```
# Find the index with the minimum value in one of the arrays you've created
np.argmin(a5)
```

```
0
```

```
# Find the indexes with the maximum values down the 1st axis (axis=1)
# of one of the arrays you created
np.argmax(a5,axis=1)
```

```
array([2, 2, 2, 2])
```

```
# Find the indexes with the minimum values across the 0th axis (axis=0)
# of one of the arrays you created
np.argmin(a5,axis=0)
```

```
array([0, 0, 1])
```

```
# Create an array of normally distributed random numbers
np.random.normal(size=100)
```

```
array([ 0.65655361, -0.97468167,  0.7870846 ,  1.15859558, -0.82068232,
        0.96337613,  0.41278093,  0.82206016,  1.89679298, -0.24538812,
       -0.75373616, -0.88951443, -0.81581028, -0.07710171,  0.34115197,
        0.2766908 ,  0.82718325,  0.01300189,  1.45353408, -0.26465683,
        2.72016917,  0.62566735, -0.85715756, -1.0708925 ,  0.48247242,
       -0.22346279,  0.71400049,  0.47323762, -0.07282891, -0.84679372,
       -1.51484722, -0.44651495,  0.85639879,  0.21409374, -1.24573878,
        0.17318093,  0.38531738, -0.88385744,  0.15372511,  0.05820872,
       -1.1429703 ,  0.35778736,  0.56078453,  1.08305124,  1.05380205,
       -1.37766937, -0.93782504,  0.51503527,  0.51378595,  0.51504769,
        3.85273149,  0.57089051,  1.13556564,  0.95400176,  0.65139125,
       -0.31526924,  0.75896922, -0.77282521, -0.23681861, -0.48536355,
        0.08187414,  2.31465857, -1.86726519,  0.68626019, -1.61271587,
       -0.47193187,  1.0889506 ,  0.06428002, -1.07774478, -0.71530371,
        0.67959775, -0.73036663,  0.21645859,  0.04557184, -0.65160035,
        2.14394409,  0.63391902, -2.02514259,  0.18645431, -0.66178646,
        0.85243333, -0.79252074, -0.11473644,  0.50498728,  0.86575519,
       -1.20029641, -0.33450124, -0.47494531, -0.65332923,  1.76545424,
        0.40498171, -1.26088395,  0.91786195,  2.1221562 ,  1.03246526,
       -1.51936997, -0.48423407,  1.26691115, -0.70766947,  0.44381943])
```

```
# Create an array with 10 evenly spaced numbers between 1 and 100
np.linspace(1,100)
```

```
array([  1.        ,   3.02040816,   5.04081633,   7.06122449,
         9.08163265,  11.10204082,  13.12244898,  15.14285714,
        17.16326531,  19.18367347,  21.20408163,  23.2244898 ,
        25.24489796,  27.26530612,  29.28571429,  31.30612245,
        33.32653061,  35.34693878,  37.36734694,  39.3877551 ,
        41.40816327,  43.42857143,  45.44897959,  47.46938776,
        49.48979592,  51.51020408,  53.53061224,  55.55102041,
        57.57142857,  59.59183673,  61.6122449 ,  63.63265306,
        65.65306122,  67.67346939,  69.69387755,  71.71428571,
        73.73469388,  75.75510204,  77.7755102 ,  79.79591837,
        81.81632653,  83.83673469,  85.85714286,  87.87755102,
        89.89795918,  91.91836735,  93.93877551,  95.95918367,
        97.97959184, 100.        ])
```

Extensions

For more exercises, check out the NumPy quickstart tutorial. A good practice would be to read through it and for the parts you find interesting, add them into the end of this notebook.

Pay particular attention to the section on broadcasting. And most importantly, get hands-on with the code as much as possible. If in dobut, run the code, see what it does.

The next place you could go is the Stack Overflow page for the top questions and answers for NumPy. Often, you'll find some of the most common and useful NumPy functions here. Don't forget to play around with the filters! You'll likely find something helpful here.

Finally, as always, remember, the best way to learn something new is to try it. And try it relentlessly. If you get interested in some kind of NumPy function, asking yourself, "I wonder if NumPy could do that?", go and find out.

✓ 0s completed at 10:52 AM

Exercise Background

This small application based coding exercise is ment to expose you to the use of the numpy library as well as give you a taste of tasks that you might be needed to perform during machine learning.

Usually, machine learning involves working on large data sets. This notebook will walk you through normalising the data and then dividing the data set into smaller subsets. It is recommended that while attempting each of the tasks visit the NumPy library to find the most appropriate function which can help you achieve the desired result. More often than not you will find the functions which you require prewritten in the library. The **numpy library** can be found here.

Without further ado, the first task is to mean normalise a data set. Mean normalising is a data transformation done to reduce the variations in the data set. For example, consider a data set which has integers between 0 and 10000. That is a lot of variation, and it becomes difficult to build ML algorithms on this data. So mean normalisation is done on such data, after the transformation, the mean of the data will be zero, and standard deviation will be 1. Even though the actual values of data will change a lot, but the overall variation is still kept intact. If the concept of normalisation feels a bit unclear dont worry all of this will be covered in the future sections of this program. For now, let's concentrate on the tasks at hand.

Task 1: Mean Normalisation:

Question 1.1 Create a 2D of random integers between 0 and 10,000 (including both 0 and 10,000) with 25000 rows and 15 columns. This will be the dataset you will use in the notebook.

```
# import NumPy into Python
import numpy as np

# Create a 25000 x 15 ndarray with random integers in the interval [0, 10000].
x = np.random.randint(0, 10000, (25000, 15))

# print the shape of X
x.shape
```

```
(25000, 15)
```

```
# print the first row of X
x[0]
```

```
array([6709, 2695, 1103, 1942, 5674, 7843, 2701, 6492, 3119, 5485, 5297,
       439, 4998, 1428, 4293])
```

Now that you created the array we will mean normalize it. The equation for normalisaing the data is given below:

$$\text{Norm\_Col}_i = \frac{\text{Col}_i - \mu_i}{\sigma_i}$$

where Col_i is the ith column of X, μ_i is average of the values in the ith column of X, and σ_i is the standard deviation of the values in the ith column of X. To put it simply, to find the new value of each element, you have to subtract the mean of respective column form that value and divide the result with the standard deviation of that columns. Now the question is, Why are these operations being done column-wise? That is because usually all the procedures in ML are done column-wise. So it will be beneficial for us to develop the habit of thinking about data column-wise.

Question 1.2 Find the mean and the standard deviation of each of the columns in the dataset. The result will be two 1D arrays with 15 elements each, representing the mean and standard deviation for each of the columns in the dataset.

```
# Average of the values in each column of X
ave_cols = np.mean(x, axis=0)

# print ave_cols
print("Average\n", ave_cols)

print()

# Standard Deviation of the values in each column of X
std_cols = np.std(x, axis=0)

# print std_cols
print("STD\n", std_cols)
```

```
Average
 [4998.106   5006.76804 5015.10524 5004.49904 4993.4756  5022.68828
 4998.55448 5006.80012 4999.00456 4984.46188 4965.977   4993.23036
 5016.59784 5011.91428 4964.96332]

STD
 [2880.66346806 2893.28632907 2882.07046591 2888.48518667 2882.31183051
 2897.98499764 2886.55822821 2899.37376759 2895.56664379 2887.45987917
 2895.52641762 2880.80146248 2887.20367159 2886.56792452 2866.33443267]
```

Question 1.3 Print the shape of each both the arrays, they should have 15 elements each.

```
# Print the shape of ave_cols
print(ave_cols.shape)

# Print the shape of std_cols
print(std_cols.shape)
```

```
(15,)
(15,)
```

Question 1.4 Now that you have mean and standard deviation calculated, it is time to apply the transformation to the dataset.

HINT The broadcast property of NumPy can make this a lot easier. You can read about it here. All you have to do is create one row of transformation values and repeat them through all the values.

```
# Mean normalize X
x_norm = (x-ave_cols)/std_cols
x_norm
```

```
array([[ 0.59392359, -0.79901115, -1.35739403, ..., -0.00644147,
        -1.24158321, -0.23443298],
       [-0.94391658, -1.01883039,  0.7122986 , ...,  0.64124404,
         0.46009162,  0.05443771],
       [-0.64919281, -0.9915258 , -0.18948365, ...,  0.20275749,
        -0.80092149,  0.50588538],
       ...,
       [-0.72417553, -0.38494912,  1.29451892, ...,  0.82758351,
        -1.72901328, -1.29885867],
       [ 1.6620803 , -1.51999061, -1.37023202, ..., -1.3693519 ,
         0.97766129, -0.64157319],
       [-1.51496558, -1.56492221,  1.48639487, ..., -1.72817661,
        -1.6150371 ,  0.03315617]])
```

```
x_norm.shape
```

```
(25000, 15)
```

Question 1.5 If the transformation has been performed correctly, the mean of elements in each column will be approximately 0. Also, the average of the **minimum** value in each column of X_norm and the average of the **maximum** value in each column of X_norm will have almost the same face value with opposite signs. Let's confirm if the transformation has happened correctly.

```
# Print the average of all the values of X_norm
print(np.mean(x_norm))

# Print the average of the minimum value in each column of X_norm
print(np.mean(np.min(x_norm, axis=0)))
```

```
# Print the average of the maximum value in each column of X_norm
print(np.mean(np.max(x_norm, axis=0)))
```

```
2.980489928935034e-17
-1.731266124227261
1.7317574123950974
```

Be mindful that the exact values might not match since the dataset was initialized using the random function.

Data Spliting

After data processing, it is a regular practice in ML to split the dataset into three datasets.

1. A Training Set
2. A Cross Validation Set
3. A Test Set

The ratios in which the data is split varies a bit from case to case. But the accepted standard 6:2:2 for train, test, and validation respectively. That is 60% for training data and so on. Again why is the data split or what is the signification of these smaller data sets? These questions are better left unanswered for now. The tanks assigned to you is to split the data in the given proportions randomly. For instance, if the data set had ten elements, this is how you would do it.

```
# We create a random permutation of integers 0 to 9
np.random.permutation(10)
```

```
array([8, 3, 7, 5, 2, 6, 1, 9, 0, 4])
```

1. training set = 8,3,7,5,2,6
2. Cross Validation Set = 1,9
3. Test Set = 0,4

Question 2.1 Similarly, create a 1D array representing the indexes of the rows in the dataset X_norm. U can use the `np.random.permutation()` function for randomising the indexes.

```
# Create a rank 1 ndarray that contains a random permutation of the row indices of `X_norm`
row_indices = np.random.permutation(25000)
```

```
array([16067, 19171, 18859, ...,  5009,  1011, 23173])
```

```
# Print the shape of row_indices
row_indices.shape
```

```
(25000,)
```

Question 2.2 Split the row indexes in the needed proportions. You can use the slicing methods you have learnt in this session to make the job easier.

```
# Make any necessary calculations.
# You can save your calculations into variables to use later.
train = row_indices[:15000]
test = row_indices[15000:20000]
val  = row_indices[20000:]
```

Question 2.3 Now make use of the indexes that you made to split the data also similarly once the data is split print the shape of each of the smaller data sets. `X_train` should have 15000 rows and 15 columns. `X_test` should have 5000 rows and 15 columns. `X_val` should have 5000 rows and 15 columns.

```
# Create a Training Set
train_set = x_norm[train]
print(train_set)

# Create a Cross Validation Set
validationset = x_norm[val]
print(validationset)
```

```
# Create a Test Set
testset = x_norm[test]
print(testset)
```

```
[[-1.00258362  0.105151    0.95379166 ...  1.64532977  1.4020407
   1.50332656]
 [-0.53706586  0.12727118  1.53670592 ...  1.64325164 -0.31834147
  -1.39305563]
 [-1.68610671 -0.08079672 -0.77794949 ... -0.69222614  1.23124964
   0.77068351]
 ...
 [ 0.74076477  0.8441031  -0.17907447 ... -1.43758401 -0.66408078
   0.54914621]
 [-0.85817244 -0.87539488 -0.5649082  ... -0.24542704 -0.08034257
  -1.25908662]
 [-1.14282909  1.71197434 -1.70679562 ... -1.04447008 -1.21733296
   0.93605151]]
[[-1.35597443 -0.54393788  0.79244931 ...  0.93495384  0.14518478
  -0.44759722]
 [-1.27057743 -0.85845912 -1.6242161  ... -1.04758728 -1.48720362
  -1.63622335]
 [ 0.60433786  1.20770348  0.18385906 ...  0.91936783 -1.20105065
  -0.39352118]
 ...
 [-1.24593034  1.58340082  0.01974093 ...  0.7971042  -0.68971676
  -1.71227867]
 [ 1.65166603  1.09364632  1.02353318 ...  1.40599785 -0.79918933
  -1.5043476 ]
 [-0.13854656 -0.01616433  1.55544245 ... -0.67560105  0.69809053
   1.27620721]]
[[ 1.54648193  1.69019979  1.43816566 ...  0.3478806   1.32132201
  -0.14442255]
 [ 1.60480183 -0.00717801 -0.14854087 ... -0.4019799  -1.10231748
   0.7661481 ]
 [ 0.99417861 -1.1211362  -0.34458049 ...  0.12171021 -0.76316038
   0.10572272]
 ...
 [-1.52364414 -0.24704366  0.044376   ...  0.9030891  -0.74376018
  -0.35933118]
 [ 0.38945681  0.12139551  1.71366204 ...  1.15211898 -0.32527011
  -0.89032295]
 [-1.57745118  1.72026941  0.55824269 ... -1.25401539  1.66359699
  -0.56656449]]
```

```
# Print the shape of X_train
print(train_set.shape)

# Print the shape of X_crossVal
print(validationset.shape)

# Print the shape of X_test
print(testset.shape)
```

```
(15000, 15)
(5000, 15)
(5000, 15)
```

Colab paid products - Cancel contracts here

✓ 0s completed at 11:13 AM