q1

February 11, 2021

[1]:
```
from timeit import default_timer
from numpy import array
from numpy.random import random
```

Part A: Returns a numpy array whose elements are the squares of a given array (construct in two ways)

[2]:
```
# a function squares each element of the original array
def square_array(original_array):

    squared_array = list(map(lambda x: x**2, original_array))
    square_array = array(squared_array)

    return squared_array
```

[3]:
```
my_array = array([1, 2, 3, 4, 5, 7, 8, 9, 10])
squared = square_array(my_array)
```

[4]:
```
squared
```

[4]:
```
[1, 4, 9, 16, 25, 49, 64, 81, 100]
```

[5]:
```
# a function that uses direct numpy syntax to construct a new array
def square_array_numpy(original_array):

    squared_array = original_array**2

    return squared_array
```

[6]:
```
my_array = array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
squared = square_array_numpy(my_array)
```

[7]:
```
squared
```

[7]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100], dtype=int32)
```

Part B: Construct a one-dimensional numpy array of length 10^4 and reports (for each of the array squaring functions) the time taken by 1000 consecutive calls to that function

[8]:
```
# a function that reports the runtime for the two above functions after 1000␣
↪consecutive calls
def report_runtimes(random_array_size, num_iterations):

    random_array = random([random_array_size])
    print(random_array, random_array.size)

    timer_start_function1 = default_timer()

    # run the function that iterates over the original array
    for _ in range(num_iterations):
        square_array(random_array)

    timer_end_function1 = default_timer()
    timer_start_function2 = default_timer()

    # run the function that uses numpy element-wise multiplication
    for _ in range(num_iterations):
        square_array_numpy(random_array)

    timer_end_function2 = default_timer()

    # use values of the default timer to report the runtimes for each function
    print("Avg. time (1 call) w/ iteration: {:.8f} seconds".
↪format((timer_end_function1 - timer_start_function1)))
    print("Avg. time (1 call) w/ numpy squaring: {:.8f} seconds".
↪format((timer_end_function2 - timer_start_function2)))
```

[9]:
```
report_runtimes(10**4, 1000)
```

```
[0.86276176 0.74465247 0.90752341 … 0.56539004 0.20987613 0.0474892 ] 10000
Avg. time (1 call) w/ iteration: 7.67086260 seconds
Avg. time (1 call) w/ numpy squaring: 0.00561270 seconds
```

q2

February 10, 2021

[1]:
```
%matplotlib inline

from numpy import mean, std, array, median, percentile
from numpy.random import random
from matplotlib.pyplot import boxplot
from math import sqrt
```

[2]:
```
x = random((1000,))
# X
```

Part A: Compute the sample's Mean and Standard Deviation Using Numpy functions

[3]:
```
sample_mean_x = mean(x)
sample_mean_x
```

[3]:
```
0.5057000380848997
```

[4]:
```
sample_std_x = std(x)
sample_std_x
```

[4]:
```
0.29539677637048345
```

Part B: Compute the standardized version of the original sample of x. We can convert the dataset to a standard normal distribution by applying the following standardization to each x_i

$$z = \frac{x - \bar{x}}{\sigma}$$

[5]:
```
def standardize(x, sample_mean, std):

    x_standardized = (x - sample_mean) / std

    return x_standardized
```

[6]:
```
x_standardized = array([standardize(x_i, sample_mean_x, sample_std_x) for x_i␣
↪in x])
# x_standardized
```

We took a normal distribution and standardized it so that it becomes a standard normal distribution. By definition of the standard normal distribution, the mean should be 0 and the standard deviation should be 1. We check in the code below:

[7]:
```
mean_standardized = mean(x_standardized)
mean_standardized
```

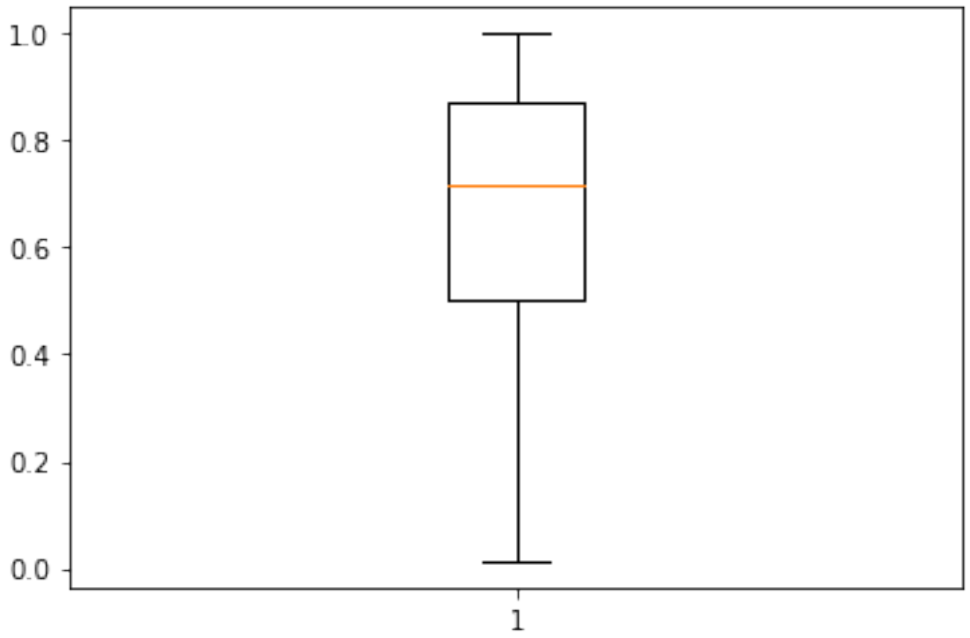
[7]: 1.8474111129762604e-16

[8]:
```
std_standardized = std(x_standardized)
std_standardized
```

[8]: 1.0

Part C: Draw a boxplot of the set of square roots of the elements of the sample x (plot with matplotlib.pyplot.boxplot)

[9]:
```
x_square_roots = array([sqrt(u) for u in x])

box = boxplot(x_square_roots)
# box
```



Part D: Using Numpy, compute the numerical locations of the three parallel line segments that form the main body of the box plot above

In other words, we need to find the 3rd quartile (topmost line), the 1st quartile (bottommost line), and the median (the line in the middle)

[10]:
```
# middlemost line -> median

median = median(x_square_roots)
median
```

[10]: 0.7152223829609285

[11]:
```
# topmost line -> 3rd quartile
# the 3rd quartile is the same as the 75th percentile

quartile3 = percentile(x_square_roots, 75)
quartile3
```

[11]: 0.8701397127134112

[12]:
```
# bottommost line -> 1st quartile
# the 1st quartile is the same as the 25th percentile

quartile3 = percentile(x_square_roots, 25)
quartile3
```

[12]: 0.5030572095323681

q3

February 10, 2021

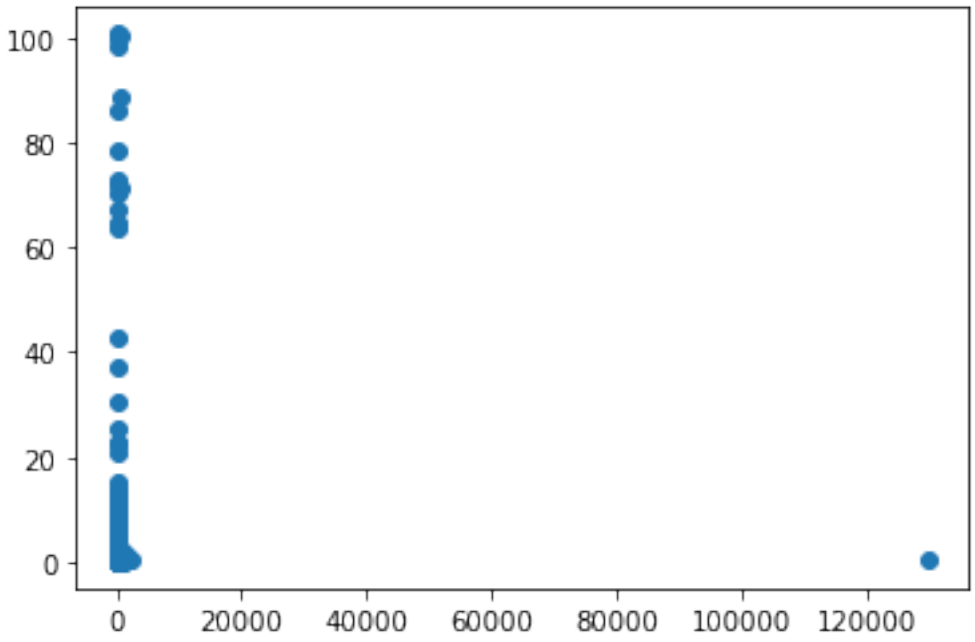
[1]:
```
%matplotlib inline

from numpy import loadtxt, mean, std, invert
from matplotlib.pyplot import scatter
```

Part A: Load the numerical portion of the data and plot an attribute space representation of the data set

[2]:
```
kepler_data = loadtxt('./kepler_selection.csv', skiprows=1, delimiter=",")
kepler_data.shape
```

[2]:
```
(9564, 2)
```

[3]:
```
attribute_space = scatter(kepler_data[:, 0], kepler_data[:, 1])
```



The resulting plot is not very useful towards understanding the distribution of the data in the space of attributes. This is because the outliers greatly distory the chart which makes it nearly

impossible to identify any real trends in the data.

Part B: In this case we define an outlier as a point (x, y) such that the corresponding z-value for the x component only satisfies $|z| > 5$

[4]:
```
def standardize(x, x_bar, std):

    standardized = (x - x_bar) / std

    return standardized
```

[5]:
```
def isOutlier(x, x_bar, stdev):

    return abs(standardize(x, x_bar, stdev)) > 5
```

[6]:
```
x = kepler_data[:, 0]
x
```

[6]:
```
array([ 9.48803557, 54.4183827 , 19.89913995, …,  1.73984941,
        0.68140161,  4.85603482])
```

[7]:
```
x_bar = mean(x)
x_bar
```

[7]:
```
75.67135842490904
```

[8]:
```
stdev = std(x)
stdev
```

[8]:
```
1334.674264645049
```

[9]:
```
outliers = [isOutlier(x_i, x_bar, stdev) for x_i in x]
# outliers
```

[10]:
```
num_outliers = sum(outliers)
num_outliers
```

[10]:
```
1
```

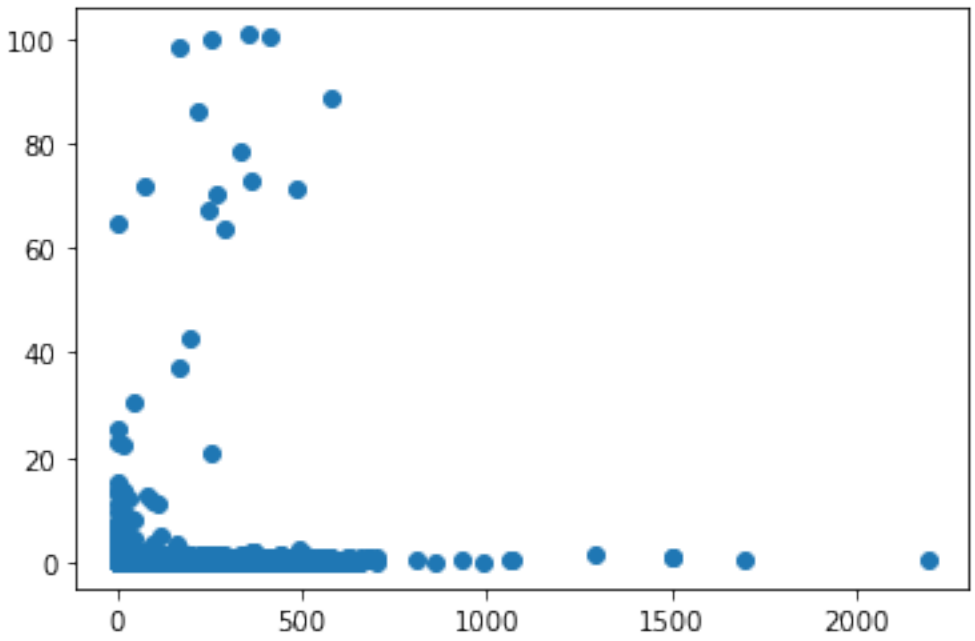
Part C: Use boolean indexing to pre-process the kepler data by keeping only the data instances that are not outliers. Generate an attribute space view of the set of non-outliers

[11]:
```
non_outliers = kepler_data[invert(outliers)]
non_outliers
```

[11]:
```
array([[9.48803557e+00, 1.46000000e-01],
       [5.44183827e+01, 5.86000000e-01],
```

```
       [1.98991399e+01, 9.69000000e-01],
       …,
       [1.73984941e+00, 4.30000000e-02],
       [6.81401611e-01, 1.47000000e-01],
       [4.85603482e+00, 1.34000000e-01]])
```

[12]:
```
attribute_space = scatter(non_outliers[:, 0], non_outliers[:, 1])
```



The results are much better for providing a good view of the data. Before, the outlier skewed how our data looked on the plot, so we could not see any real trends in the data we actually cared about. By removing it, however, we are able to better see the patterns/trends that we want to see in the relevant data.