CSCI2291 Homework 2

Jack Moffatt

February 3, 2022

Problem 1

(a) We construct the vectorized function and iterative approaches each under their own function:

```
1    import numpy as np
2
3    def square_of_array(a):
4        return a ** 0.5
5
6    def square_of_array_2(a):
7        for i, term in enumerate(np.nditer(a)):
8            a[(i)] = term ** 0.5
9        return a
```

The iterative approach is likely more time complex then the vectorized NumPy operation, but this hypothesis will be tested in part b.

(b) Now, we will time the runtimes of each of these function calls for a a NumPy array of length 10^4. We can use the $/textttttimeit.default_timer$ function to time the our runtime. Our code reads:

```
1    import timeit
2
3    a = np.random.rand(10 ** 4)
4
5    start1 = timeit.default_timer()
6    for i in range(1001):
7        a1 = square_of_array(a)
8    end1 = timeit.default_timer()
9    start2 = timeit.default_timer()
10   for i in range(1001):
11       a2 = square_of_array_2(a)
12   end2 = timeit.default_timer()
13
14   time1 = end1 - start1
15   time2 = end2 - start2
```

```
16      time_ratio = time1 / time2
17
18      print(f"{'Vectorized Form Time:' :<30}{round(time1, 3) :<6.3f}")
19      print(f"{'Iterative Form Time:' :<30}{round(time2, 3) :<6.3f}")
20      print(f"{'Vectorized/Iterative Ratio:' :<30}{round(time_ratio, 3) :<6.3f}")
```

After running our code, we have the output

```
1       >>> Vectorized Form Time:         0.010
2       >>> Iterative Form Time:          7.079
3       >>> Vectorized/Iterative Ratio:   0.001
```

As you can see from the ratio in the third printed line, the iterative approach is exponentially slower than the vectorized NumPy approach. This shows the value in knowing the libraries you are using to effectively and effectively develop programs.

Problem 2

As a prerequisite for this problem, we write the following lines

```
1       import matplotlib
2       import numpy as np
3       from matplotlib.pylab import show
4
5       x = np.random.normal(3, 1, 1000)
```

Now we may begin with the problems.

(a) We see that we can easily use the standard NumPy functions for mean and standard deviation. We have

```
mean = round(np.mean(x), 3)
std = round(np.std(x), 3)
print(f"{'Mean:'}{mean :>6}{'STD:' :>15}{std :>6}")
>>> Mean: 3.021          STD: 0.981
```

(b) Likewise, if we apply the formula for standardization, we can then reapply the same mean and standard deviation functions to verify that we correctly standardized the data.

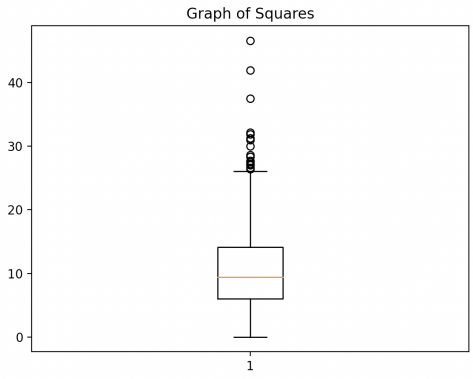
```
xStand = (x - np.mean(x)) / (np.std(x))
meanStand = round(np.mean(xStand), 3)
stdStand = round(np.std(xStand), 3)
print(f"{'Mean:'}{meanStand :>6}{'STD:' :>15}{stdStand :>6}")
>>> Mean:   0.0          STD:   1.0
```

As the mean is 0 and the standard deviation is 1, we know that we correctly standardized the data.

(c) We also can visualize this data using `matplotlib.pyplot.boxplot`. Specifically, we will graph the sqaured power of each value in the array. Our code reads

```
xSquares = x ** 2
matplotlib.pyplot.boxplot(xSquares)
matplotlib.pyplot.title("Graph of Squares")
show()
```

We used the vectorized format to sqaure the values in the array, as we experimented with in part **a**, and then we create a boxplot of the squared array. Upon running our script, `matplotlib` outputs the following box plot:



(d) Finally, we can compute the precise numerical statistics that our boxplot helps us visualize. We will compute the first quartile, median, and thhird quartile. Using the python script

```
median = round(np.percentile(xSquares, 50), 3)
firstQ = round(np.percentile(xSquares, 25), 3)
thirdQ = round(np.percentile(xSquares, 75), 3)
print(f"{'First Quartile' :<20}{'Median' :<20}{'Third Quartile' :<20}")
print(f"{firstQ :<20}{median :<20}{thirdQ :<20}")
```

we can get an output giving us our numerical statistics:

```
>>> First Quartile    Median          Third Quartile
>>> 6.039             9.426           14.113
```

Problem 3

Again, as a prerequisite to the remainder of this problem, we head our python file with the following statements

```
import numpy as np
import matplotlib.pyplot
from matplotlib.pyplot import show
```

Now we may begin our work.

(a) We may begin by loading our data set using `np.loadtxt`:

```
D = np.loadtxt('python_files/datasets/data.csv', delimiter = ',', skiprows=1, usecols=tuple(
  range(2, 13)))
```

Unpacking this function call, we see our first parameter is a path to the CSV file. The delimiter paramter lets NumPy know what character separates our data values. The skiprows parameter tells NumPy to not import the data from the first row, as this is the title of each column and is not a numerical value. Finally, the usecols parameter uses the standard python `tuple()` and `range()` functions to tell NumPy to use the data from the columns 2 to 12.

Now, we may print the shape of our array, `D` by using

```
print(D.shape)
```

which gives us the output

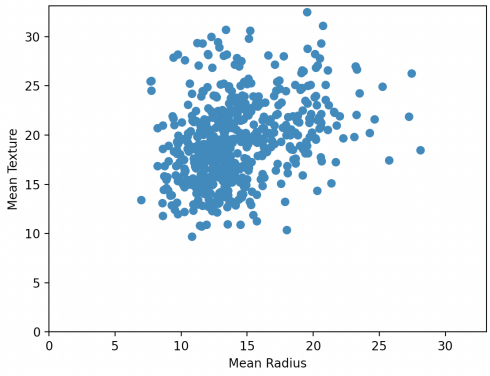
```
>>> (569, 11)
```

telling us that we have 11 attributes for each data point and 569 data points in total.

(b) It will be helpful to visualize this data set. In particular, we are concerned with the Mean Radius and Mean Texture attributes. In our array, these are the columns indexed 1 and 2 respectively. Using `matplotlib.pyplot.scatter()`, we can graph these two columns on a scatterplot. We use the python code

```
1    matplotlib.pyplot.scatter(D[:,1], D[:,2])
2    matplotlib.pyplot.xlabel("Mean Radius")
3    matplotlib.pyplot.ylabel("Mean Texture")
4    matplotlib.pyplot.xlim((0, max(D[:,1] + 5)))
5    matplotlib.pyplot.ylim((0, max(D[:,1] + 5)))
6    show()
```

Calling `D[:,1]` as the first parameter uses the second column of D as our x-axis. Likewise, calling `D[:,2]` as the second parameter uses the third column of D as the y-axis. Then we label our axes and set the scale of the axes for a more helpful visualization. And finally we use `show()` to see the figure

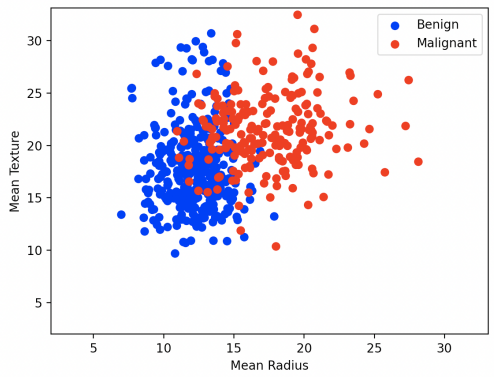


(c) While the above scatterplot is useful, it would be more helpful to be able to visualize the benign vs. malignant datapoints in seperate colors, so we could attempt to make inferences about which attribute may have more of an effect on predicting the severity. To do this, we must use boolean indexing to create seperate Numpy arrays for the benign and malignant data sets. Our python script:

```
1    meanRadBenign = D[D[:,0] == 0, 1]
2    meanRadMalignant = D[D[:,0] == 1, 1]
```

```
3     meanTextBenign = D[D[:,0] == 0, 2]
4     meanTextMalignant = D[D[:,0] == 1, 2]
5     benign = matplotlib.pyplot.scatter(meanRadBenign, meanTextBenign, c=['blue'])
6     malignant = matplotlib.pyplot.scatter(meanRadMalignant, meanTextMalignant, c=['red'])
7     matplotlib.pyplot.xlabel("Mean Radius")
8     matplotlib.pyplot.ylabel("Mean Texture")
9     matplotlib.pyplot.xlim((min(D[:,1]) - 5, max(D[:,1] + 5)))
10    matplotlib.pyplot.ylim((min(D[:,1]) - 5, max(D[:,1] + 5)))
11    matplotlib.pyplot.legend([benign, malignant], ["Benign", "Malignant"])
12    show()
```

The first for lines initilize NumPy arrays for our four datasets, whcih are sorted by their condition (benign or malignant). Next, we declare two scatter plot instances whichh follow the same index as in part **b**, using the radius attribute for the x-axis and the y-axis for the texture attribute. Additionally, we added the c parameter, to declare the color of our data points. In the benign scatter plot, our datapoints are blue, while in the malignant plot, our data points are red. Then we label the axes and set the scale, and finally we view our figure



(d) Based on the scatterplots generated in parts **b** and **c**, it seems as though it would be possible to make predictions about a patients conditions based on radius and texture measurements.

In most cases, with a mean radius less than 15 and mean texture less than 20, I feel it would be a safe guess to give a benign diagnosis. On the other hand, in instances where the mean radius is greater than 15 and the mean texture is greater than 20, I believe it would be safe to give a malignant diagnosis.

In the edge cases where only one of these conditions is satisfied, but not the other, more analysis of additional attributes could be helpful.

If I had to pick one attribute to more accurately predict diagnosis, I would likely select radius, as there seems to me a more dichotomous division across the x-axis than across the y-axis.

Problem 4

(a) Let us take a look at the python script and then unpack it step by step:

```
1   import numpy as np
2
3   D = np.loadtxt('python_files/datasets/data.csv', delimiter = ',', skiprows=1, usecols=tuple(
      range(2, 13)))
4   desc = D[:,[1,2,3,4,5,6,7,8,9,10]]
5   meanDesc = np.mean(desc, axis=0)
6
7   for size in range(0, 6):
8       N = 10 * (2 ** size) #sample size
9       sample = np.empty((0,10))
10      normSamp = []
11      for _ in range(10001):
12          sample = desc[np.random.choice(desc.shape[0], size=N, replace=True), :]
13          meanSamp = np.mean(sample, axis=0)
14          dist = abs(meanSamp - meanDesc)
15          normSamp.append(np.linalg.norm(dist))
16      normSamp = np.array(normSamp)
17      print(f"({N}, {np.mean(normSamp)})")
```

As in question **3**, we begin by loading our dataset. Then, we create a second array, `desc`, which contains only the descriptive attributes and compute the mean across the x-axis, which produces 1D array of length 10, containing the mean value of each descriptive attribute. These values will be used later in our loops.

Our outer loop while iterate over our values of `p`. For each of these values it will compute an `N` sample size and initilize an empty sample 1D array of length 10. Additionally, we will initialize an empty list to hold norm values of each sample.

Then, our inner loop will iterate 10,000 times. With each iteration, we will create an NumPy array, `sample`, which holds our `N` randomly selected rows. We will take the mean of this sample and then take the

norm value of the difference of the sample mean and the mean of the total dataset. Note the use of the vectorized subtraction in line 14 to easily calculate the distance.

Finally, after our inner loop completes, we have a list, `normSamp` of length 10,000, which we quickly convert to a 1D NumPy array so that we can take the mean. Finally, we print the results of each iteration in a tuple of the form

(N, mean of norm values)

After running this script, we get an output of

```
1    >>> (10, 88.96267114841969)
2    >>> (20, 62.6722197836738)
3    >>> (40, 44.631940662004176)
4    >>> (80, 31.083888414534048)
5    >>> (160, 22.16534548078403)
6    >>> (320, 15.679423991364414)
```

(b) As the size of our sample grows, the mean vector decreases. This suggests that as we take larger and larger sample sizes, the samples we select become more and more accurate relative relative to the entire dataset.

This makes sense, because as we select larger and larger samples of data, the chances that we can outweigh the effects of outliers in our samples become much higher. As a result, with a larger sample size, the data is closer to the whole data set.

(c) In addition, we can actually qualitatively analyze the relationship between differences in our sample means. Using the following equation

$$\text{norm for sample N } = (\frac{N}{10})^p * (\text{norm for N} = 10)$$

We can try to solve for p by plugging in values

$$89 = (\frac{10}{10})^p * 89 \qquad \implies p = -.49 \qquad (1)$$

$$63 = (\frac{20}{10})^p * 89 \qquad \implies p = -.51 \qquad (2)$$

$$44 = (\frac{40}{10})^p * 89 \qquad \implies p = -.51 \qquad (3)$$

$$31 = (\frac{80}{10})^p * 89 \qquad \implies p = -.50 \qquad (4)$$

$$22 = (\frac{160}{10})^p * 89 \qquad \implies p = -.50 \qquad (5)$$

$$16 = (\frac{320}{10})^p * 89 \qquad \implies p = -.50 \qquad (6)$$

We can see that if multiply N by 4, (looking at lines (1) and (3)) our norm value is divided by 2.

From this data we can see that our value for p is $-\frac{1}{2}$.