

Brendan Sherman

Prof. Alvarez

CSCI 2291

January 24, 2022

Homework 2 Master Writeup

1. **A)** Write a Python function that takes a numpy array, *a*, as its input, and that constructs and returns another numpy array, the elements of which are the square roots of the corresponding elements of *a*. The function should use the “vectorized” (array-level) numpy syntax *a**0.5* to construct the new array. Write a second function with the same input-output specification that uses iteration to construct the new array element by element, via either a loop or list comprehension

Source Code:

```
import numpy as np

def problem1vectorized(arr):
    return arr * .5

def problem1iterative(arr):
    return [n**2 for n in arr]
```

Explanation:

As outlined in the problem, the first function (*problem1vectorized*) uses vectorized syntax, made possible with numpy, to construct the new array of square roots. To accomplish this, all that is necessary is returning (*arr **.5*), as with a numpy array this denotes element wise operation. The second function (*problem1iterative*), uses list comprehension to accomplish the same task iteratively, accessing every element within *arr* individually and calculating the square root of each, storing these in the new array to be returned. Both of these functions use different approaches to accomplish the same task, returning an array containing square roots of all values stored in the given array (*arr*).

- B)** Write a Python program that uses *numpy.random.random* to construct a one-dimensional numpy array of length 10^4 , and that reports, for each of the two array square root functions in the preceding part, the total time taken by 1000 consecutive calls to that function (use a loop to call each function). Run the program, report your results.

Source Code:

```
import numpy as np
import timeit

#time comparison between iterative and vectorized
approaches
```

```

def problem1B():
    sample = np.random.random((10**4))

    start = timeit.default_timer()
    for i in range (0, 1000):
        problemliterative(sample)
    iterative_time = timeit.default_timer() - start

    start = timeit.default_timer()
    for i in range (0, 1000):
        problemlvectorized(sample)
    vectorized_time = timeit.default_timer() - start

    ratio = iterative_time / vectorized_time

    #rounds all values after calculating exact ratio
    iterative_time = round(iterative_time, 3)
    vectorized_time = round(vectorized_time, 3)
    ratio = round(ratio, 3)

    print("Total time for 1000 calls (iterative approach):"
    + str(iterative_time))
    print("Total time for 1000 calls (vectorized
    approach):" + str(vectorized_time))
    print("Ratio of iterative time to vectorized time: " +
    str(ratio))

def main():
    problem1B()

```

Output:

```

Total time for 1000 calls (iterative approach): 2.593
Total time for 1000 calls (vectorized approach):0.01
Ratio of iterative time to vectorized time: 269.034

```

Explanation:

As outlined by the problem, the code pasted above produces an output which compares the time taken for 1000 calls to both square root functions (vectorized and iterative). Using *numpy.random.random((10**4))*, a 1-dimensional array storing 10^4 random values is constructed. I then use two for loops to call each function 1000 times, enclosing each loop with calls to *timeit.default_timer()* to calculate the time taken for each loop to run. Finally, I divide the iterative time by the vectorized time, calculating the ratio of iterated time to vectorized time. These results are then rounded and printed, generating the above output and suggesting that vectorized array operations, made possible by numpy, are significantly more time-efficient than iterative approaches.

2. Use `numpy.random.normal` to generate a random sample, `x`, of 1000 numbers according to a normal distribution with location 3 and scale 1. Solve all of the following problems for the same sample. Include the text of your Python code in each case, together with screen shots of the actual results.

A) Compute the sample's mean and standard deviation, using numpy functions. Report the results to three digits after the decimal point, using `round`. Include the code used to generate the sample

Source Code:

```
import numpy as np

x = np.random.normal(3, 1, 1000) #loc 3, scale 1, size 1000

def problem2A():

    std = round(np.std(x, ddof=1), 3)

    mean = round(np.mean(x), 3)

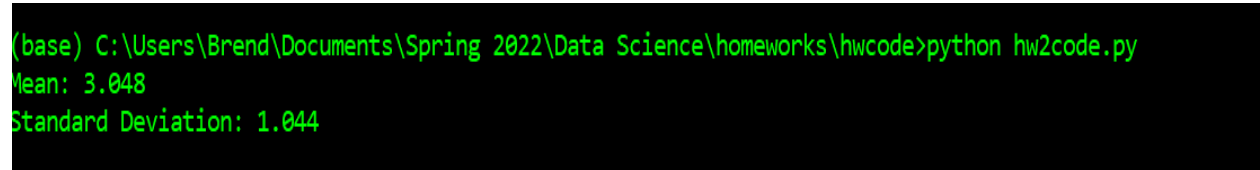
    print("Mean: " + str(mean))

    print("Standard Deviation: " + str(std))

def main():

    problem2A()
```

Output:



```
(base) C:\Users\Brend\Documents\Spring 2022\Data Science\homeworks\hwcode>python hw2code.py
Mean: 3.048
Standard Deviation: 1.044
```

Explanation:

As outlined in the problem, the code above outputs the mean and standard deviation of a random sample generated according to the normal distribution. To create the distribution, I use `np.random.normal(3, 1, 1000)`, which specifies generating a sample of size 1000, scale 1, and location 3 according to the normal distribution. I then use the built-in numpy functions `mean` and `std`, which when passed a one-dimensional array, i.e. a sample, calculate the mean and standard deviation of that sample. As discussed in class, including the parameter `ddof=1` for the call to `std()` generates a more accurate result. Finally, I round and print these values, which as expected have a mean close to the center of the distribution (3) and a standard deviation close to the scale (1).

B) Without modifying the original sample, `x`, compute its standardized version, `xStand`, as well as the mean and standard deviation of `xStand`. Report the results to three digits after the decimal point, as before. Explain.

Source Code:

```
import numpy as np
```

```

x = np.random.normal(3, 1, 1000) #loc 3, scale 1, size 1000
def problem2B():
    xStand = (x - np.mean(x)) / np.std(x, ddof=1)
    meanStand = round(np.mean(xStand), 3)
    stdStand = round(np.std(xStand, ddof=1), 3)
    print("Standardized Mean: " + str(meanStand))
    print("Standardized Standard Deviation: " + str(stdStand))
def main():
    problem2B()

```

Output:

```

(base) C:\Users\Brend\Documents\Spring 2022\Data Science\homeworks\hwcode>python hw2code.py
Standardized Mean: 0.0
Standardized Standard Deviation: 1.0

```

Explanation:

The code above outputs the mean and standard deviation of the standardized version of the array from part A, *xStand*. As with part A, I use *np.random.normal(3, 1, 1000)* to generate the sample. To compute the standardized version (wherein each value represents the number of standard deviations from the mean for the original value), I divide the difference between each value in *x* and the sample mean by the standard deviation. This is accomplished with the statement *xStand = (x - np.mean(x)) / np.std(x)*, which uses element-wise operations to create the standardized array. Finally, I use numpy's *mean* and *std* functions to calculate the mean and standard deviation for the standardized version of the sample, then round and print them. As expected, the output indicates that the standardized sample has a mean of 0 and a standard deviation of 1 (true for any standardized data).

C) Use *matplotlib.pyplot.boxplot* to draw a box plot of the set of squares of the elements of the sample, *x*, that is, of the array *[u² | u is a value in x]*.

Source Code:

```

import numpy as np
from matplotlib.pyplot import boxplot, show

x = np.random.normal(3, 1, 1000) #loc 3, scale 1, size 1000
def problem2C():

```

```

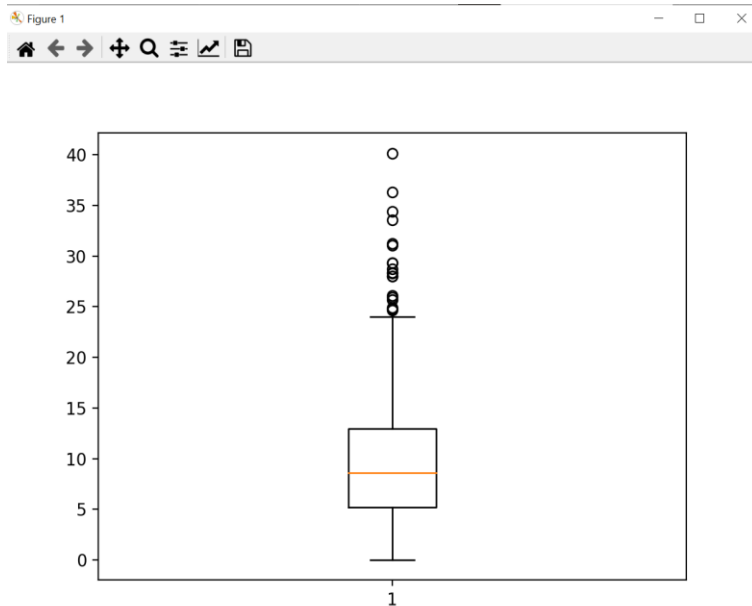
    boxplot(x**2)

    show()

def main():
    problem2C()

```

Output:



Explanation:

The above code, using matplotlib, generates the boxplot above, which represents the squares of the random sample x . First, as in the previous problems x is randomly populated according to the normal distribution. I then call `matplotlib.pyplot.boxplot(x**2)`, which generates the boxplot above, using the squares of each element in x (as denoted using numpy element-wise syntax). Finally, the call to `show()` displays the window above when the program is run.

D) Using suitable numpy functions, compute the precise numerical locations of the three parallel line segments that form the main body of the box plot in the preceding part. Explain.

Source Code:

```

import numpy as np

def problem2D():
    squares = x**2

    median = np.percentile(squares, 50) #value of middle line (median)
    first_q = np.percentile(squares, 25) #value of bottom line (Q1)
    third_q = np.percentile(squares, 75) #value of top line (Q3)

```

```

print("Bottom line (Q1): " + str(round(first_q, 3)))
print("Middle line (Median): " + str(round(median, 3)))
print("Top line (Q3): " + str(round(third_q, 3)))

def main():
    problem2D()

```

Output:

```

(base) C:\Users\Brend\Documents\Spring 2022\Data Science\homeworks\hwcode>python hw2code.py
Bottom line (Q1): 5.425
Middle line (Median): 9.016
Top line (Q3): 13.777

```

Explanation:

The above code calculates the numerical location of the three horizontal line segments making up the box plot from part C, which as we learned represent the first, second (median), and third quartiles of the sample. Numpy provides a percentile function, which when called on a given one-dimensional array calculates the given percentile. Therefore, as the quartiles are simply the 25th, 50th, and 75th, percentiles, we can use this function to calculate the location of each of the lines. Therefore, when called on x^{**2} (the sample used in the boxplot), these functions provide the locations of each of the lines from the plot. I then round and print these values to the output, yielding the above results.

3. Download the Wisconsin Breast Cancer Data Set (version 2) from the following URL:

<https://www.kaggle.com/uciml/breast-cancer-wisconsin-data/version/2>. That data set contains several hundred instances of breast cancer diagnostic imaging measurements, together with a corresponding diagnosis (benign or malignant) in each case. The file itself is in comma-separated value (CSV) format.

A) Load the numerical columns of the data set into a numpy array, using the function `numpy.loadtxt`. Read the documentation carefully in order to determine how to coax the `loadtxt` function into doing what you want it to do. Print the shape of the resulting array. Copy your source code into your writeup, including all needed import statements, the code that prints the array shape, and the shape itself.

Code:

```

import numpy as np

def problem3A():
    global data #allows global access to data for following problems
    data = np.loadtxt("data.csv", delimiter=",", skiprows=1)
    print("Numpy array shape: " + str(data.shape))

def main():
    problem3A()

```

Output:

```
(base) C:\Users\Brend\Documents\Spring 2022\Data Science\homeworks\hwcode>python hw2code.py  
Numpy array shape: (569, 32)
```

Explanation:

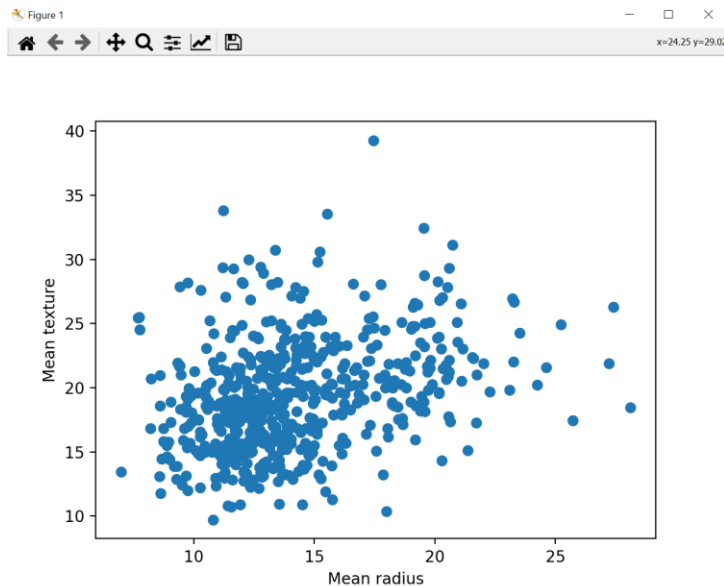
The code above accomplishes the task of loading the breast cancer dataset, originally a .csv file, into a representative numpy array. First, I added the “data.csv” file to the project directory, allowing easy access. I then modified the file in excel, using a formula to transform the “diagnosis” attribute from string to numerical representation, as numpy arrays require all data to be of the same type. I then call `np.loadtxt("data.csv", delimiter=",", skiprows=1)`, which returns an array representing the dataset. As each value in the file is separated by a comma, the `delimiter` parameter is set to “,”, allowing numpy to distinguish between values. As the first row contains attribute titles, which are strings, I specify `skiprows=1`, which means that only the numerical data is included in the array. I finally print the shape of this array, which outputs **(569, 32)**, indicating that there are 569 rows and 32 columns. This shape confirms the data was successfully loaded, as the dataset has 569 instances and 32 attributes.

B) Use `matplotlib.pyplot.scatter` to draw a scatter plot of the data, with the mean radius and mean texture attributes as coordinates. Each axis should be labeled with the name of the corresponding attribute. Read the documentation. Include your code and scatter plot in your writeup.

Code:

```
import numpy as np  
  
from matplotlib.pyplot import scatter, show, xlabel, ylabel  
  
def problem3B():  
    scatter(data[:, 2], data[:, 3])  
    xlabel("Mean radius")  
    ylabel("Mean texture")  
    show()  
  
def main():  
    problem3A() #Loads dataset  
    problem3B()
```

Output:



Explanation:

The code above generates the attached plot, which visualizes each instance within the dataset by plotting its mean radius and mean texture attributes. To create this scatter plot, I use `matplotlib.pyplot.scatter()`, passing column 2 (mean radius) as the horizontal axis and column 3 (mean texture) as the vertical axis. The numpy syntax for accessing a given attribute is `data[:, n]`, which specifies including all rows (instances) but only column n. Using this syntax, I pass the necessary columns directly to `scatter()` to define the axes. I then use matplotlib functions `xlabel()` and `ylabel()` to appropriately label the axes, finally calling `show()` to display the generated plot, yielding the figure above.

C) Now use the same scatter function to draw a grouped scatter plot, using the same pair of attributes as before. To do this, invoke scatter twice, once for each class (each diagnosis), before calling `show()`. For each class, you can use Boolean indexing to isolate the rows corresponding to that class, as in `diagnosis==0`, for example, though you'll need to extract the diagnosis column from the data set by referring to its column number. Plot the two classes in different colors. Read the documentation. Include source code and grouped scatter plot in your writeup.

Code:

```
import numpy as np

from matplotlib.pyplot import scatter, show, xlabel, ylabel

def problem3C():
    diagnosis = data[:, 1]

    #divides data into malignant and benign diagnoses
    malig = data[diagnosis == 1]
    benign = data[diagnosis == 0]

    #creates and outputs grouped scatter plot
```



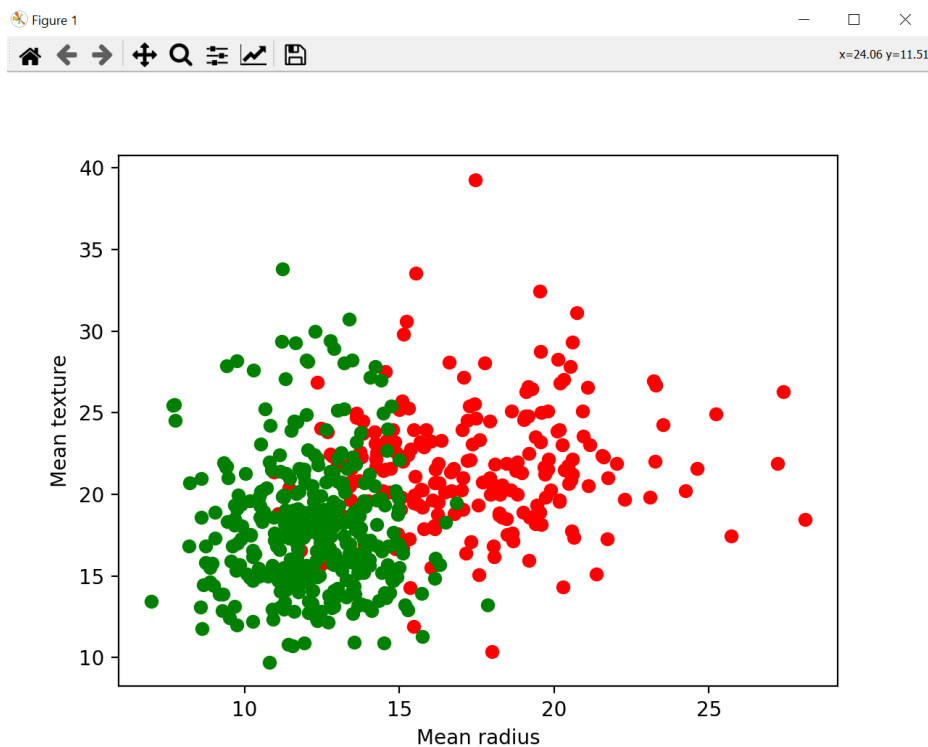
```

scatter(malig[:, 2], malig[:, 3], c="r")
scatter(benign[:, 2], benign[:, 3], c="g")
xlabel("Mean radius")
ylabel("Mean texture")
show()

def main():
    problem3A() #loads dataset
    problem3C()

```

Output:



Explanation:

As outlined in the problem, the code above generates a plot which visualizes each instance within the dataset by plotting its mean radius and mean texture, grouped by the diagnosis attribute. I first isolate the “diagnosis” column into a 1 dimensional numpy array using `diagnosis = data[:, 1]`, which accesses all rows (instances) of the data but only column 1 (the diagnosis). I then use this column with Boolean indexing to group the original dataset by diagnosis, generating two arrays containing all the benign instances and all the malignant instances respectively. To fetch the malignant doses I use `malig = data[diagnosis == 1]`, which specifies including all rows (instances) where the diagnosis attribute is equal to 1 (indicating malignance). For benign, I do the same thing, indexing by `diagnosis == 0`. I then call `scatter()` twice to plot the mean radius and mean texture for both groups of data, as in part B.

The only difference between the two `scatter()` calls is the `c` parameter, denoting color, which is equal to “r” for the malignant group and “g” for the benign. In this way, instances with a malignant diagnosis are plotted as red, benign as green. I finally call `show()`, generating the grouped scatter plot above.

D) Based on the scatter plots that you generated in this task, does it seem that it would be possible to differentiate between the two diagnoses (in most cases, not all) based only on the mean radius and mean texture attributes? If you needed to select only one of these two attributes in order to differentiate between classes, which one would it be? Answer these questions concisely and justify your answers in terms of concrete observations concerning the scatter plots. Be specific.

Answer: Using the generated scatter plots, particularly the grouped scatter plot from part C, it *does* seem viable to differentiate between diagnoses using these two attributes (mean texture and mean radius). As visible in the grouped scatter plot, the points representing malignant diagnoses (red) and benign diagnoses (green) appear to have little overlap, with both colored groups of data seeming to occupy distinct locations on the plot. As these groups appear to be separated horizontally (benign datum generally to the left and malignant datum generally to the right), the attribute on the horizontal axis (**Mean radius**) would appear to better differentiate between the two classes. While not an absolute, the general line of distinction (separating the two groups) appears to be about $x=15$, suggesting that tumors with a mean radius attribute greater than 15 are malignant, less than 15 benign. Therefore, by grouping and visualizing the data in this way, predictive thinking (differentiating diagnosis based on mean radius) is enabled.

4. Continue working with the breast cancer data set from the preceding task. You will study, empirically, the effect of sampling on the sample mean values of the descriptive attributes. Even though the full data set is itself a sample from a larger population, in this task we will view the data set as playing the role of the population; you will work with smaller samples consisting of subsets of selected instances of the data set.

A) Write a program in Python that will repeat the following computations 10,000 times for each of the six sample sizes $N = 10 \cdot 2^p$, where $p = 0, 1, 2, 3, 4, 5$. Your program should then print the value of N , together with the corresponding mean of the 10,000 observed norm values. It should, therefore, print six pairs of values in all (one pair per value of N). Run your program and report the results

Code:

```
import numpy as np

def problem4A():
    sample_sizes = [10 * (2**p) for p in [0, 1, 2, 3, 4, 5]]
    data_means = np.mean(data[:, 2:], axis=0) #population mean vector
    avg_norms = []

    for n in sample_sizes:
        total_norms = 0
        for i in range(10000):
            rows = np.random.choice(569, size=n)
            sample = data[rows, 2:] #generates sample from given rows
            #Sample mean vector
            sample_means = np.mean(sample, axis=0)
            total_norms += np.linalg.norm(sample_means - data_means)

        #store the mean norm value for each sample size
        avg_norms.append(total_norms / n)

    #Print output
    for i in range(len(sample_sizes)):
        avg_norms[i] = round(avg_norms[i], 3)
        print("N: " + str(sample_sizes[i]) + ", Mean norm value: " +
              str(avg_norms[i]))

def main():
    problem3A() #loads dataset
    problem4A()
```

Output:

```
N: 10, Mean norm value: 170738.611
N: 20, Mean norm value: 61433.103
N: 40, Mean norm value: 21706.041
N: 80, Mean norm value: 7593.031
N: 160, Mean norm value: 2682.642
N: 320, Mean norm value: 951.931
```

Explanation:

The above code produces the output attached, illustrating the mean value of geometric distances between the dataset (population) mean vector and sample mean vector for 10000 samples of each given size. To accomplish this, first I compute the population mean vector with `np.mean()` by taking the mean value of each descriptive attribute within the dataset. The parameter “`axis=0`” denotes taking the mean value of each column (attribute). I then iterate through an array of sample sizes (following the formula described in the problem). For each sample size, I generate 10000 random samples using `np.random.choice` to randomly select N values between 0 and 569, such that each value represents a row in the dataset. I then use indexing to populate a sample array with the chosen rows, calling `np.mean()` with `axis=0` on this array to generate the sample mean vector. Finally, for each sample the geometric distance between the sample mean vector and population mean vector is computed using `np.linalg.norm()`, with the average geometric distance for each sample size eventually being stored in an array. Last, I iterate through this array, printing the respective sample sizes and mean distances, generating the output attached above.

B) Based on your results, how, qualitatively, does the relationship between the sample mean vector and the mean vector of the full data set change with the size of the sample? Discuss, pointing to specific evidence from your program’s results.

Answer: Viewing the results of part A, it is clear that as the sample size increases, the geometric distance between the sample mean vector and population mean vector decreases. More specifically, each time the sample size doubles, it appears that the geometric distance decreases by a factor of roughly one third. This is a trend that holds for each case in the output. In other words, you can gather that as the sample size increases, the sample and population mean vectors become “closer”, or more similar.

C) Building on the preceding part, can you describe the relationship between the sample means quantitatively, by expressing the norm of their difference algebraically, in the following form?

norm for sample size $N = f(N) * (\text{norm for sample size } 10)$

*Assume that $f(N)$ is of the form $(N/10)^p * (\text{norm for sample size } 10)$*

Answer:

Computing ratios

$170738.611 \text{ (norm for } N=10) / 21706.041 \text{ (norm for } N=40) = \mathbf{7.8659489}$

$21706.041 \text{ (norm for } N=40) / 2682.642 \text{ (norm for } N=160) = \mathbf{8.09129}$

$7593.031 \text{ (norm for } N=80) / 951.931 \text{ (norm for } N=320) = \mathbf{7.976451}$

Therefore: the norm decreases by a factor of ~8 when the sample size is quadrupled.

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

$$\begin{aligned} \rightarrow \text{norm for sample size } 40 &= (4)^p * 170738.611 \\ \therefore 4^p &= \frac{1}{8} \text{ (because we expect a decrease by a factor of roughly } 1/8) \end{aligned}$$

$$\therefore p = -\left(\frac{3}{2}\right)$$

We can then express the relationship as:

norm for sample size $N = (N/10)^{-1.5} * 170738.611$