TAYLOR'S UNIVERSITY

Wisdom • Integrity • Excellence

ITS66904 Big Data Technologies ASSIGNMENT Cover Sheet

HAND OUT DATE:	**8th February 2022**
HAND IN DATE:	**20th February 2022**
WEIGHTAGE:	**15%**
INTAKE:	**Semester January 2022**

Instructions to student:

- This is an individual assignment.
- Complete this cover sheet and attach it to your assignment – this should be your first page!

Student declaration:	
I declare that: ✦ ***I understand what is meant by plagiarism*** ✦ ***The implication of plagiarism has been explained to me by our lecturer This assignment is my own work.***	
Names	**Student ID**
Alister Animesh Baroi	0340938

Objectives / Module Learning Outcomes (MLO 3)

The objective of this assessment is to enable the students to:

Table of Contents

1.0 Introduction

In this individual assignment, I have decided to use a dataset archive from the internet to find an ideal dataset to create an **Artificial Neural Network** model to predict the outcome of breast cancer diagnosis, based on data extracted from X-ray scans of breast cancer patients (that are inputted into the dataset). The dataset comes with some dirty and noisy data, such as irrelevant columns, outliers, unusable data types, and more.

For this assignment project, I will be using the **Anaconda** environment, which is a data science and machine learning environment, consisting of all the tools and python libraries required to do data analysis and machine learning. For the IDE, I'll use the **Jupyter Notebook**, which is the ideal IDE for this project, and it is included in the Anaconda environment. The python libraries that will be used in this project are as follows:

- **Numpy**: For fast and efficient mathematical computation
- **Pandas**: For data analysis and data cleaning
- **Matplotlib**: For data visualization
- **Seaborn**: For advanced data visualization
- **Scikit Learn:** For doing machine learning tasks (splitting data into training/testing part, scaling data, showing prediction accuracy and score)
- **Keras**: For making the deep learning model

We will start off this project by first opening the Jupiter notebook and importing the basic, necessary libraries. Note, more required libraries will be imported later on, but the basic essential ones must be imported first. Figure 1 below shows the importing of the libraries.

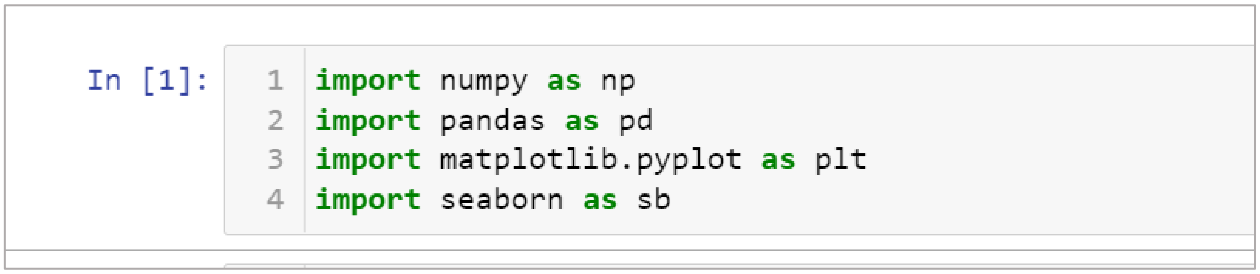
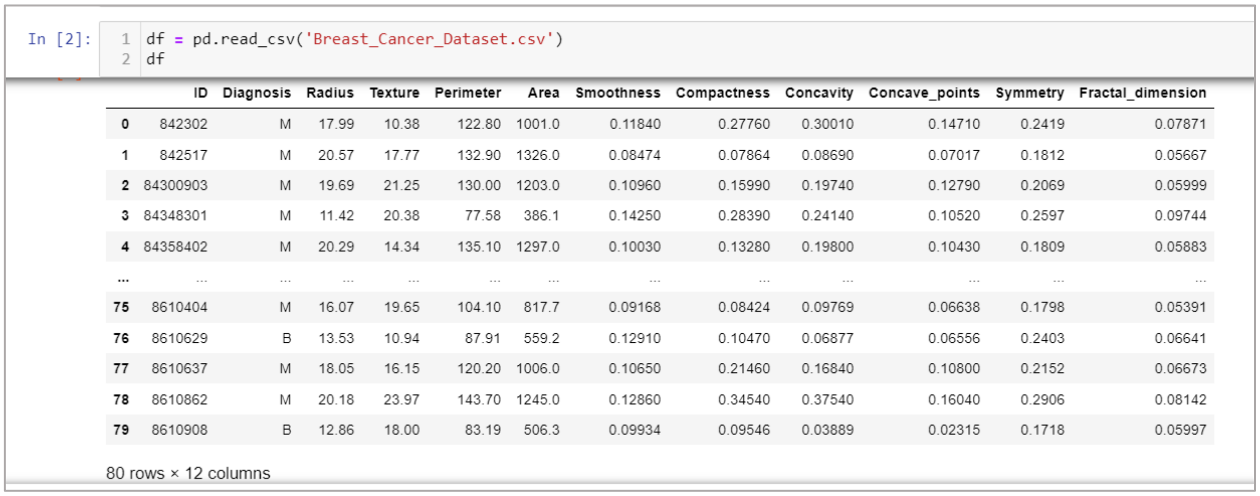

```
In [1]:
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
```

Figure 1: Importing Libraries

2.0 Gathering & Analyzing Data

2.1 Importing Dataset & View Basic Info

After importing the libraries, now we will first import the dataset using **Pandas**, and see the dataset. The figure below illustrates the process.



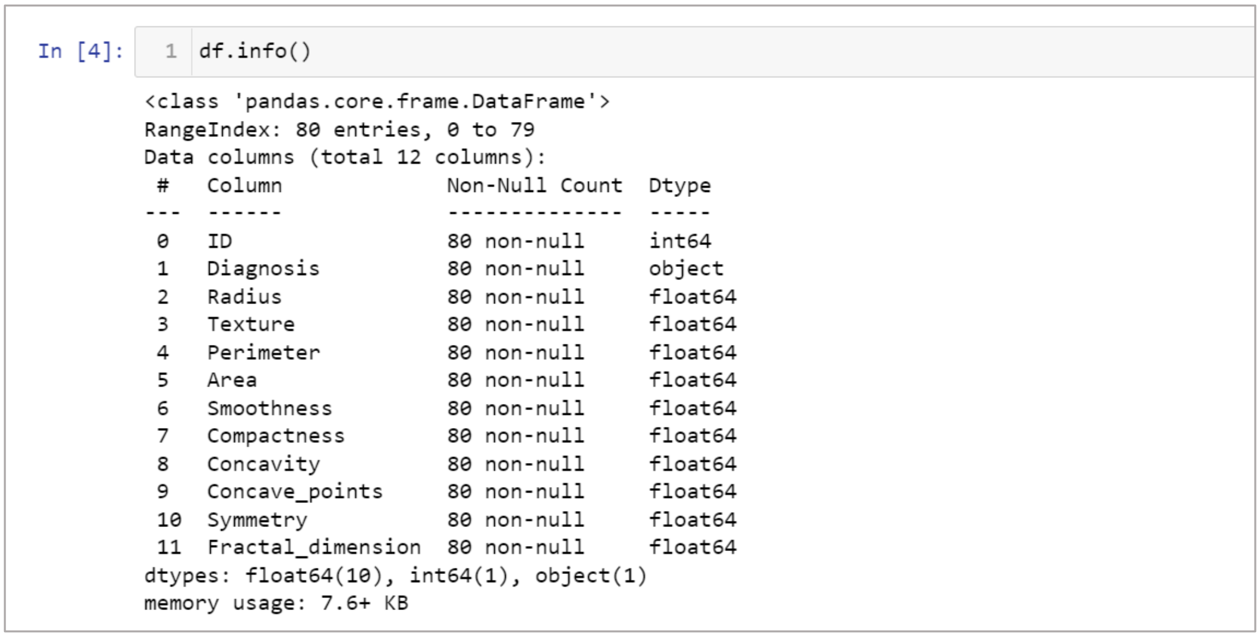
```
In [2]: df = pd.read_csv('Breast_Cancer_Dataset.csv')
        df
```

	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.07871
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.05667
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.05999
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	0.09744
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	0.05883
...	...	...	...	...	...	...	...	...	...	...	...	...
75	8610404	M	16.07	19.65	104.10	817.7	0.09168	0.08424	0.09769	0.06638	0.1798	0.05391
76	8610629	B	13.53	10.94	87.91	559.2	0.12910	0.10470	0.06877	0.06556	0.2403	0.06641
77	8610637	M	18.05	16.15	120.20	1006.0	0.10650	0.21460	0.16840	0.10800	0.2152	0.06673
78	8610862	M	20.18	23.97	143.70	1245.0	0.12860	0.34540	0.37540	0.16040	0.2906	0.08142
79	8610908	B	12.86	18.00	83.19	506.3	0.09934	0.09546	0.03889	0.02315	0.1718	0.05997

80 rows × 12 columns

Figure 2: Importing dataset using Pandas

As we can see from Figure 2, the dataset contains 12 columns, and 80 rows. Figure 3 shows more details about the dataset.

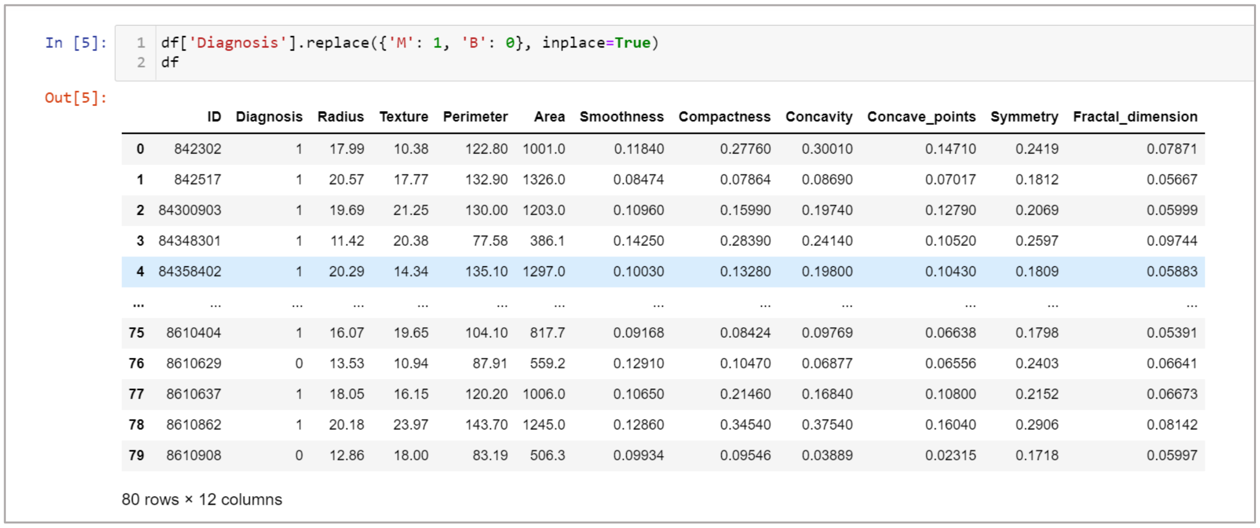


```
In [4]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 80 entries, 0 to 79
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   ID                 80 non-null     int64
 1   Diagnosis          80 non-null     object
 2   Radius             80 non-null     float64
 3   Texture            80 non-null     float64
 4   Perimeter          80 non-null     float64
 5   Area               80 non-null     float64
 6   Smoothness         80 non-null     float64
 7   Compactness        80 non-null     float64
 8   Concavity          80 non-null     float64
 9   Concave_points     80 non-null     float64
 10  Symmetry           80 non-null     float64
 11  Fractal_dimension  80 non-null     float64
dtypes: float64(10), int64(1), object(1)
memory usage: 7.6+ KB
```

Figure 3: Dataset Description

2.2 Converting Qualitative Data to Quantitative Data

From Figure 3, we can see that the “Diagnosis” column’s data type is object type. Since the diagnosis column is dependent on the other columns, it’s the output column, and thus needs to be quantitative data, integer to be precise. So, the Figure 4 below shows the Diagnosis column being converted into quantitative data column by replacing the “**M**” (Malignant) with 1, and “**B**” (Benign) with 0.



In [5]:
```
df['Diagnosis'].replace({'M': 1, 'B': 0}, inplace=True)
df
```

Out[5]:

	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	1	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.07871
1	842517	1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.05667
2	84300903	1	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.05999
3	84348301	1	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	0.09744
4	84358402	1	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	0.05883
...	...	...	...	...	...	...	...	...	...	...	...	...
75	8610404	1	16.07	19.65	104.10	817.7	0.09168	0.08424	0.09769	0.06638	0.1798	0.05391
76	8610629	0	13.53	10.94	87.91	559.2	0.12910	0.10470	0.06877	0.06556	0.2403	0.06641
77	8610637	1	18.05	16.15	120.20	1006.0	0.10650	0.21460	0.16840	0.10800	0.2152	0.06673
78	8610862	1	20.18	23.97	143.70	1245.0	0.12860	0.34540	0.37540	0.16040	0.2906	0.08142
79	8610908	0	12.86	18.00	83.19	506.3	0.09934	0.09546	0.03889	0.02315	0.1718	0.05997

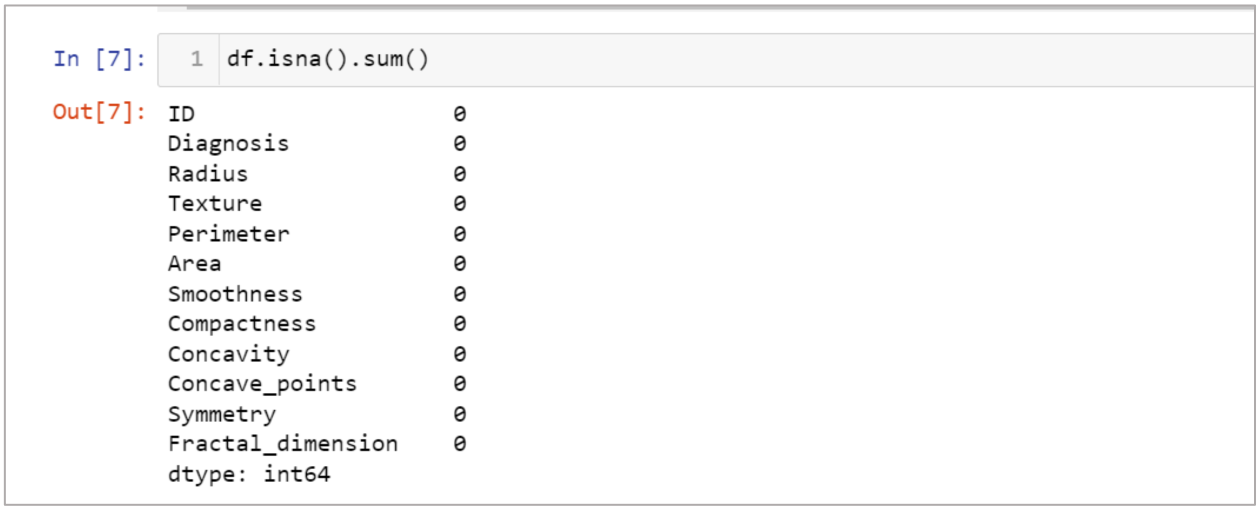
80 rows × 12 columns

Figure 4: Converting 'Diagnosis' to Quantitative Data

After converting the data type, we can see from the image above, that now the dataset shows 1s and 0s for the Diagnosis column.

2.3 Cleaning Missing Values

Figure 5 shows that there are no missing values for any column, so we don’t need to fix it.



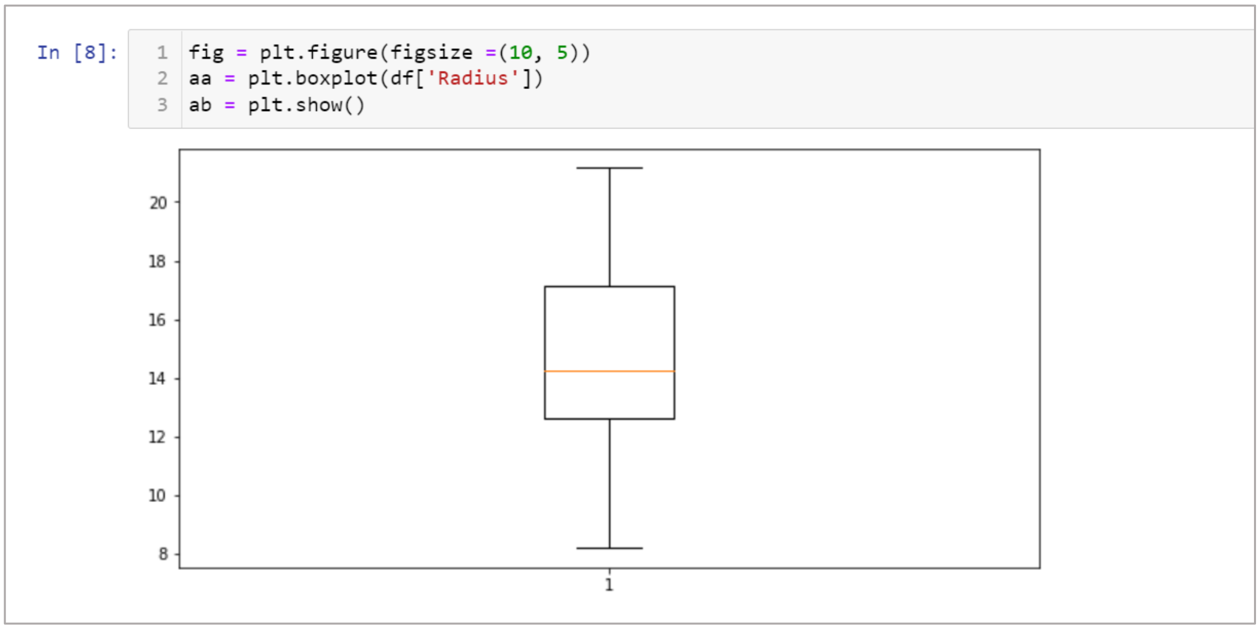
In [7]:
```
df.isna().sum()
```

Out[7]:
```
ID                   0
Diagnosis            0
Radius               0
Texture              0
Perimeter            0
Area                 0
Smoothness           0
Compactness          0
Concavity            0
Concave_points       0
Symmetry             0
Fractal_dimension    0
dtype: int64
```

Figure 5: Seeing Total Number of Missing Values of each Column

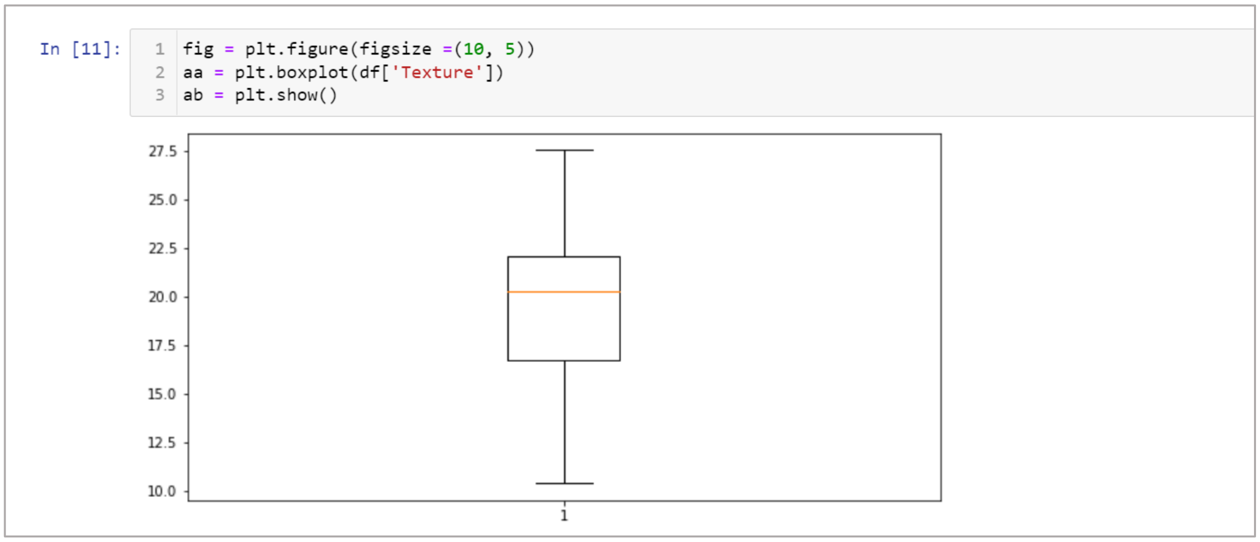
2.4 Cleaning Outliers

Now we will look for outliers for each of the columns (except ID & Diagnosis column). We will first plot boxplots of each column to easily visualize and look for outliers. If any outliers are found in the boxplots of any column, then we will clean that data by replacing the value with mean value of that column.



```
In [8]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Radius'])
ab = plt.show()
```

Figure 6: Boxplot of the "Radius" Column



```
In [11]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Texture'])
ab = plt.show()
```

Figure 7: Boxplot of the "Texture" Column

```
In [14]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Perimeter'])
ab = plt.show()
```

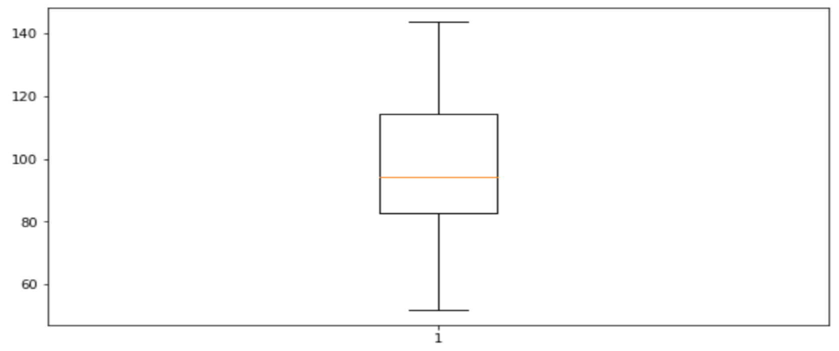


Figure 8: Boxplot of the "Perimeter" Column

```
In [18]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Area'])
ab = plt.show()
```

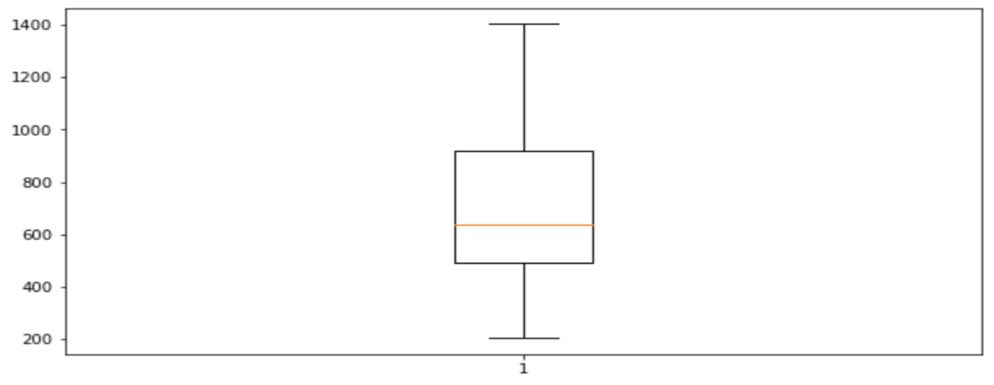


Figure 9: Boxplot of the "Area" Column

```
In [21]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Smoothness'])
ab = plt.show()
```

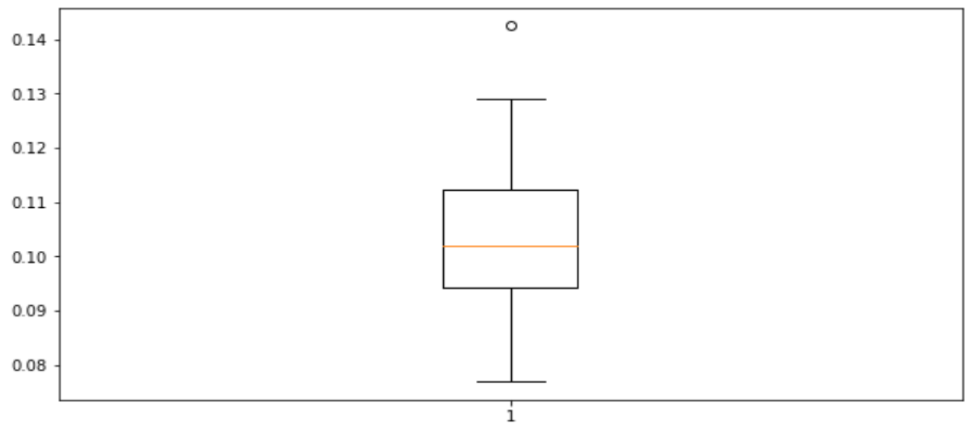
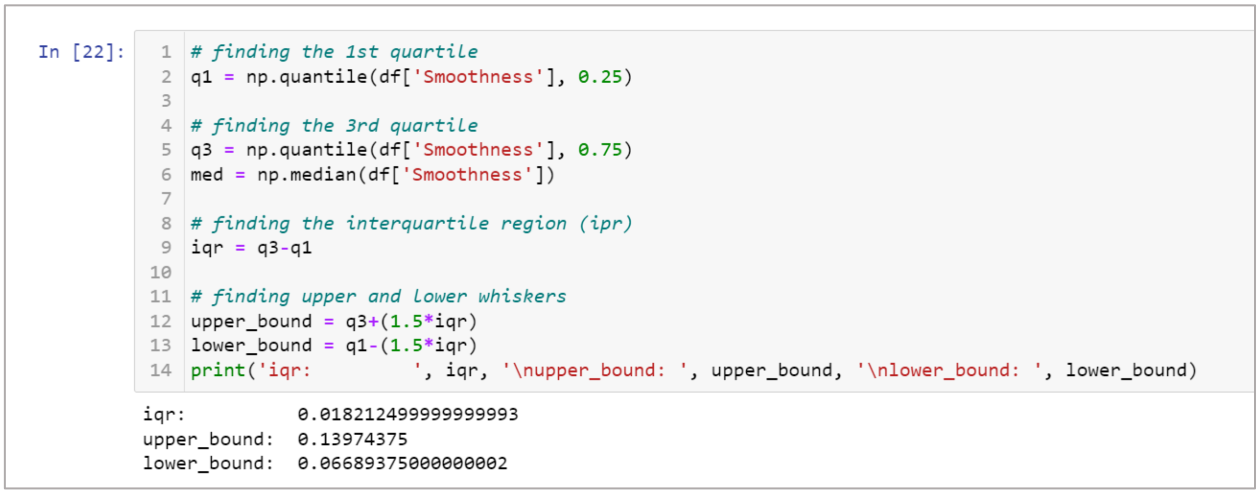


Figure 10: Boxplot of the "Smoothness" Column

On Figure 10, we can see that the **Smoothness** column has 1 outlier. So now, we have to find the upper and the lower bounds (whiskers) to find the exact value of the outlier. Figure 11 below shows how the upper and the lower bounds are calculated.

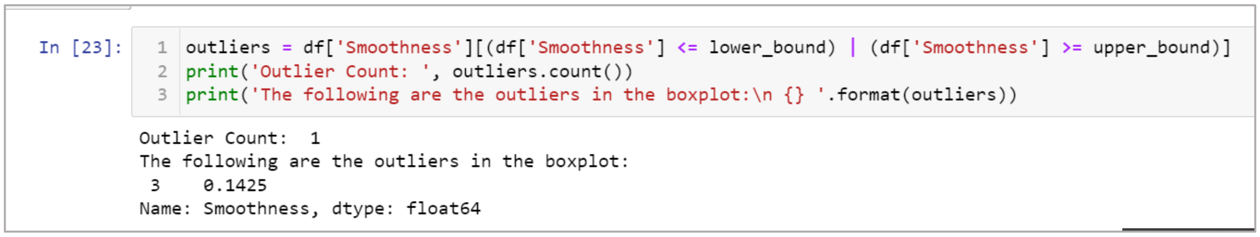


```
In [22]:
# finding the 1st quartile
q1 = np.quantile(df['Smoothness'], 0.25)

# finding the 3rd quartile
q3 = np.quantile(df['Smoothness'], 0.75)
med = np.median(df['Smoothness'])

# finding the interquartile region (ipr)
iqr = q3-q1

# finding upper and lower whiskers
upper_bound = q3+(1.5*iqr)
lower_bound = q1-(1.5*iqr)
print('iqr:         ', iqr, '\nupper_bound: ', upper_bound, '\nlower_bound: ', lower_bound)
```

```
iqr:          0.018212499999999993
upper_bound:  0.13974375
lower_bound:  0.06689375000000002
```

Figure 11: Finding the Upper & Lower Bounds of "Smoothness"

Now that we have calculated the upper bound and the lower bound of the “Smoothness” column, we can use it to find the exact outliers, and their values. Figure 12 below shows how it is done.

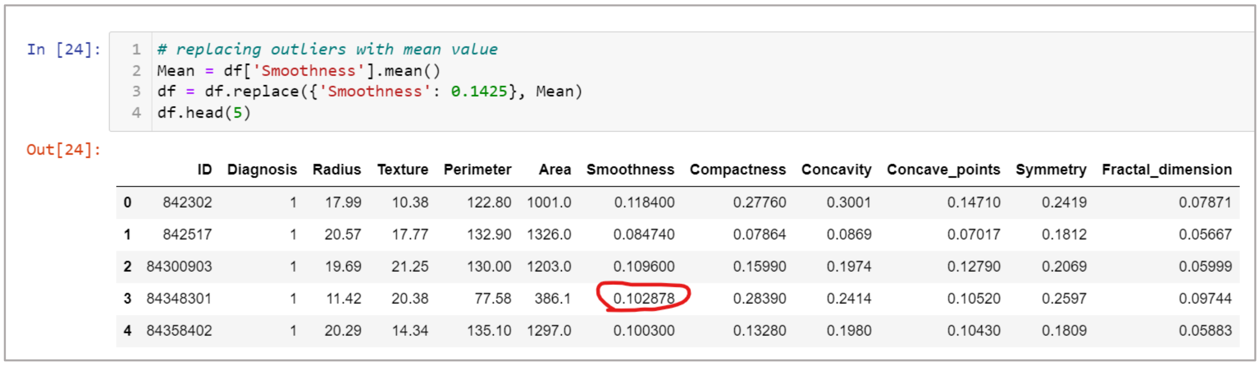


```
In [23]:
outliers = df['Smoothness'][(df['Smoothness'] <= lower_bound) | (df['Smoothness'] >= upper_bound)]
print('Outlier Count: ', outliers.count())
print('The following are the outliers in the boxplot:\n {} '.format(outliers))
```

```
Outlier Count:  1
The following are the outliers in the boxplot:
 3    0.1425
Name: Smoothness, dtype: float64
```

Figure 12: Finding outlier using the Bounds of "Smoothness"

Now that we found the exact outlier, we can replace it with the mean value of the “Smoothness” column. It is shown below on Figure 13, showing the replaced (mean) value, circled in red.



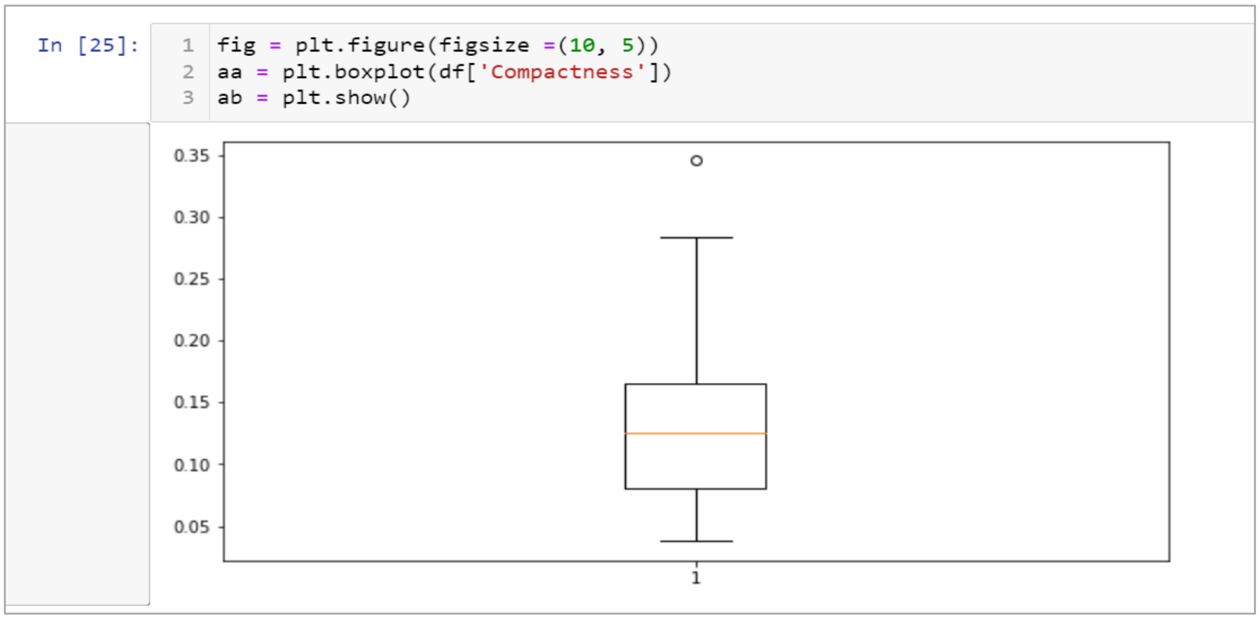
```
In [24]:
# replacing outliers with mean value
Mean = df['Smoothness'].mean()
df = df.replace({'Smoothness': 0.1425}, Mean)
df.head(5)
```

Out[24]:

	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	1	17.99	10.38	122.80	1001.0	0.118400	0.27760	0.3001	0.14710	0.2419	0.07871
1	842517	1	20.57	17.77	132.90	1326.0	0.084740	0.07864	0.0869	0.07017	0.1812	0.05667
2	84300903	1	19.69	21.25	130.00	1203.0	0.109600	0.15990	0.1974	0.12790	0.2069	0.05999
3	84348301	1	11.42	20.38	77.58	386.1	0.102878	0.28390	0.2414	0.10520	0.2597	0.09744
4	84358402	1	20.29	14.34	135.10	1297.0	0.100300	0.13280	0.1980	0.10430	0.1809	0.05883

Figure 13: Replacing Outlier of "Smoothness" with Mean Value

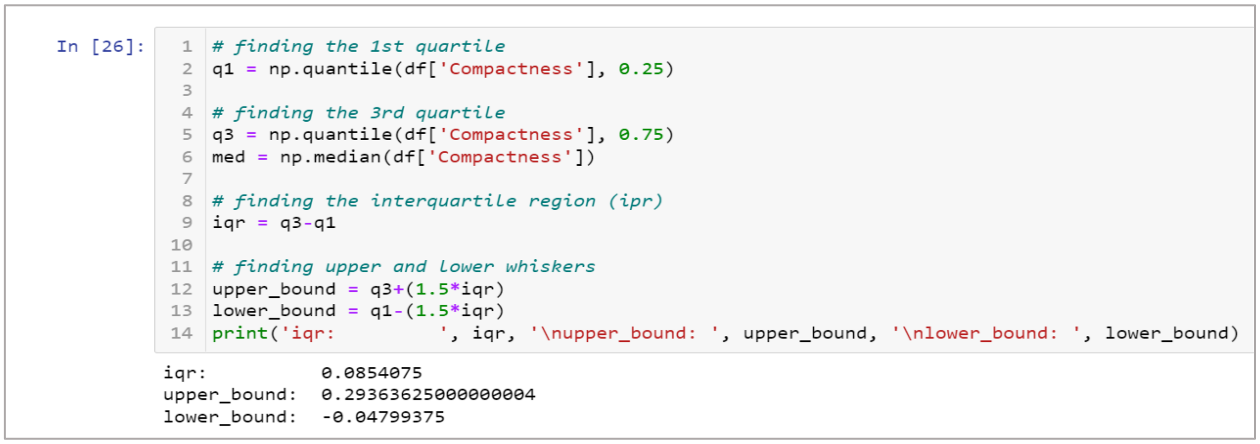
Now we will continue finding outliers of the remaining columns using the boxplot.



```
In [25]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Compactness'])
ab = plt.show()
```

Figure 14: Boxplot of the "Compactness" Column

Figure 14 shows that the “Compactness” column also has an outlier. Now we will find the bounds of this column (shown in Figure 15).

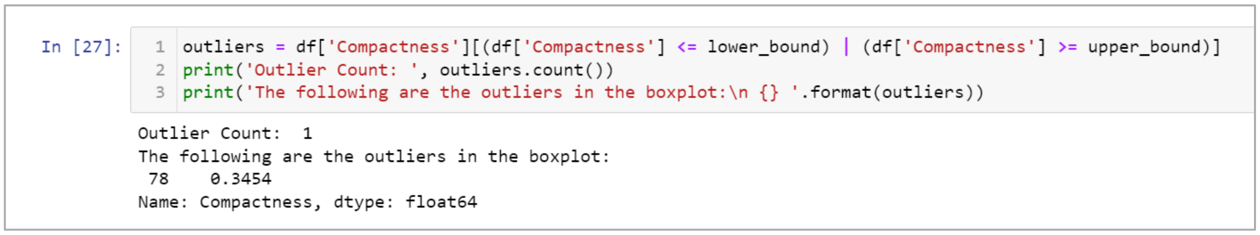


```
In [26]:
# finding the 1st quartile
q1 = np.quantile(df['Compactness'], 0.25)

# finding the 3rd quartile
q3 = np.quantile(df['Compactness'], 0.75)
med = np.median(df['Compactness'])

# finding the interquartile region (ipr)
iqr = q3-q1

# finding upper and lower whiskers
upper_bound = q3+(1.5*iqr)
lower_bound = q1-(1.5*iqr)
print('iqr:          ', iqr, '\nupper_bound: ', upper_bound, '\nlower_bound: ', lower_bound)
```

```
iqr:          0.0854075
upper_bound:  0.29363625000000004
lower_bound:  -0.04799375
```

Figure 15: Finding the Upper & Lower Bounds of "Compactness"

Now that we have calculated the upper bound and the lower bound of “Compactness” column, we can use it to find the exact outliers, and their values. Figure 12 below shows how it is done.

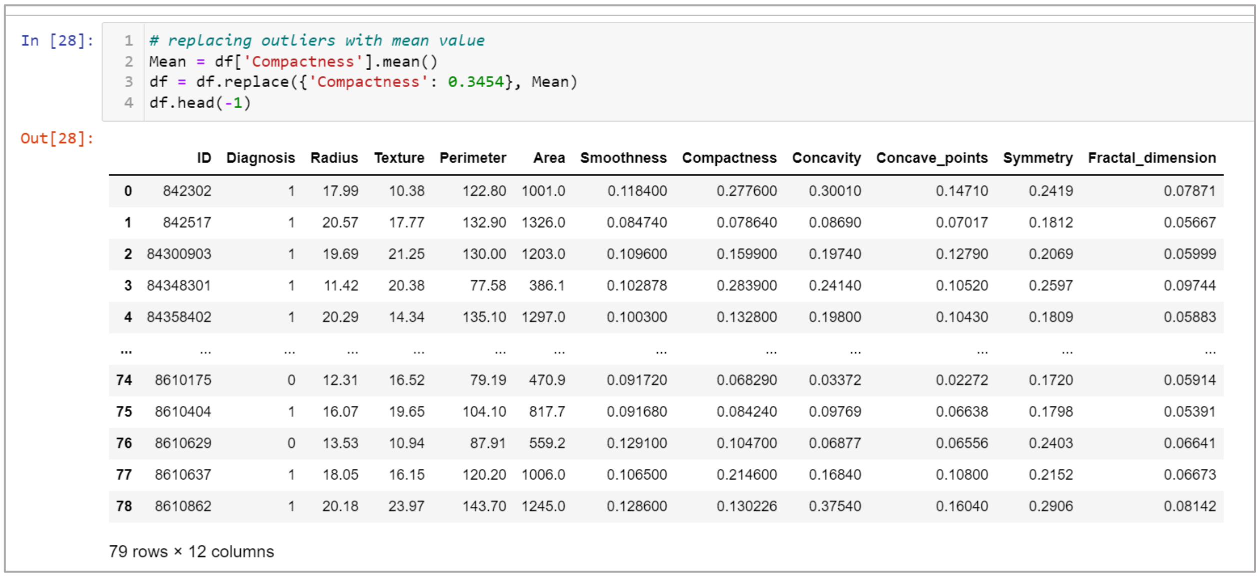


```
In [27]:
outliers = df['Compactness'][(df['Compactness'] <= lower_bound) | (df['Compactness'] >= upper_bound)]
print('Outlier Count: ', outliers.count())
print('The following are the outliers in the boxplot:\n {} '.format(outliers))
```

```
Outlier Count:  1
The following are the outliers in the boxplot:
 78    0.3454
Name: Compactness, dtype: float64
```

Figure 16: Finding Outlier using the Bounds of "Compactness"

Now that we found the exact outlier, we can replace it with the mean value of the “Compactness” column. It is shown below on Figure 17.



```
In [28]:
# replacing outliers with mean value
Mean = df['Compactness'].mean()
df = df.replace({'Compactness': 0.3454}, Mean)
df.head(-1)
```

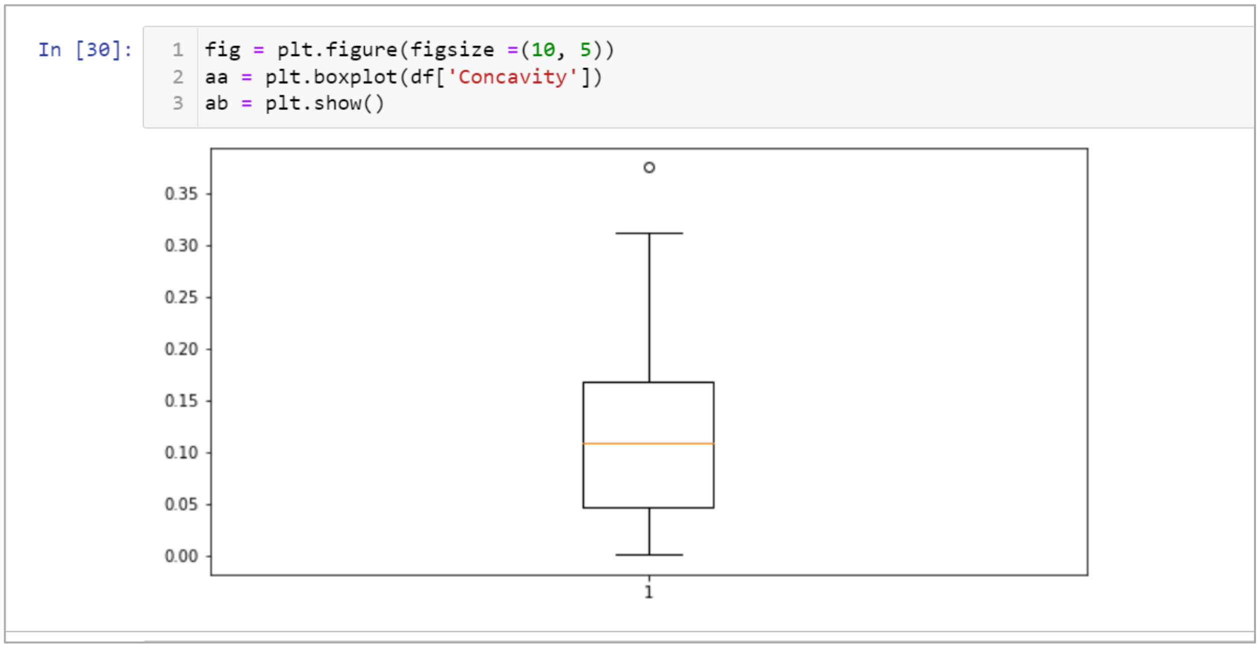
Out[28]:

	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	1	17.99	10.38	122.80	1001.0	0.118400	0.277600	0.30010	0.14710	0.2419	0.07871
1	842517	1	20.57	17.77	132.90	1326.0	0.084740	0.078640	0.08690	0.07017	0.1812	0.05667
2	84300903	1	19.69	21.25	130.00	1203.0	0.109600	0.159900	0.19740	0.12790	0.2069	0.05999
3	84348301	1	11.42	20.38	77.58	386.1	0.102878	0.283900	0.24140	0.10520	0.2597	0.09744
4	84358402	1	20.29	14.34	135.10	1297.0	0.100300	0.132800	0.19800	0.10430	0.1809	0.05883
...	...	...	...	...	...	...	...	...	...	...	...	...
74	8610175	0	12.31	16.52	79.19	470.9	0.091720	0.068290	0.03372	0.02272	0.1720	0.05914
75	8610404	1	16.07	19.65	104.10	817.7	0.091680	0.084240	0.09769	0.06638	0.1798	0.05391
76	8610629	0	13.53	10.94	87.91	559.2	0.129100	0.104700	0.06877	0.06556	0.2403	0.06641
77	8610637	1	18.05	16.15	120.20	1006.0	0.106500	0.214600	0.16840	0.10800	0.2152	0.06673
78	8610862	1	20.18	23.97	143.70	1245.0	0.128600	0.130226	0.37540	0.16040	0.2906	0.08142

79 rows × 12 columns

Figure 17: Replacing Outlier of "Compactness" with Mean Value

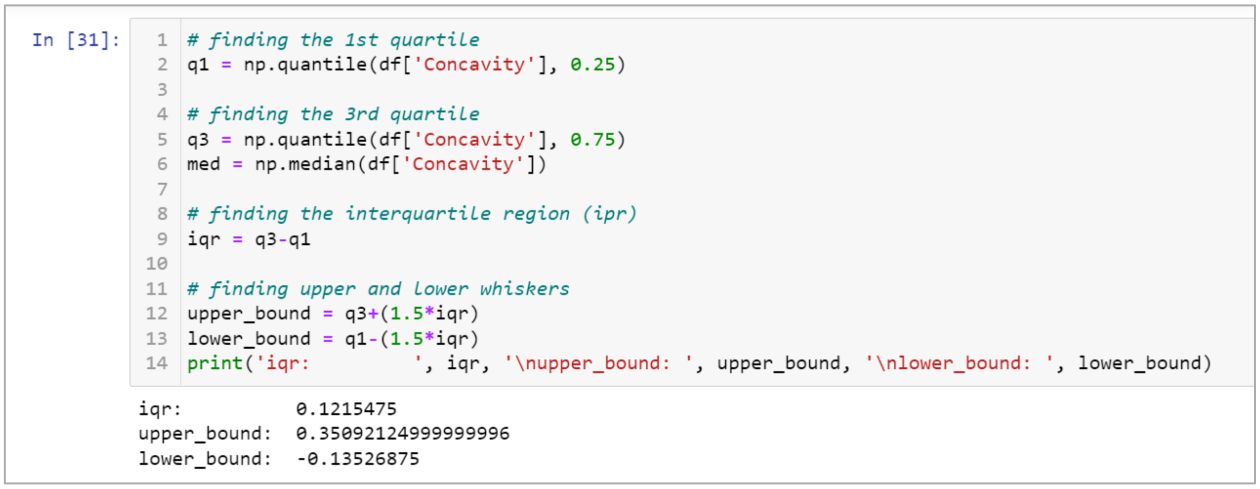
Again, now we will continue finding outliers of the remaining columns using the boxplot.



```
In [30]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Concavity'])
ab = plt.show()
```

Figure 18: Boxplot of the "Concavity" Column

Figure 18 shows that the “Concavity” column also has an outlier. Now we will find the bounds of this column (shown in Figure 19).

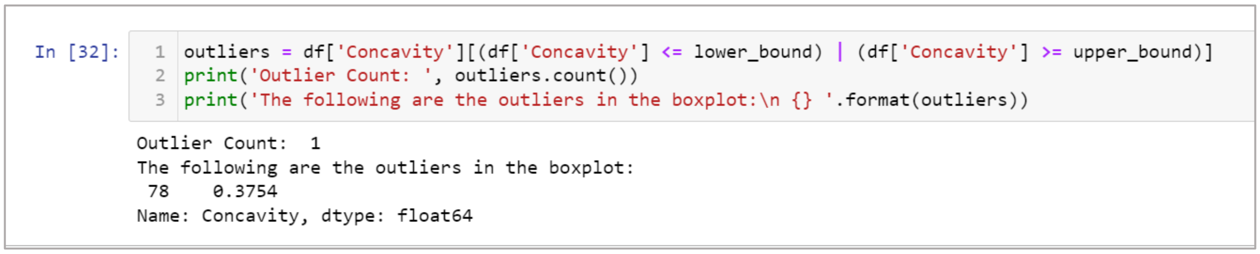


```
In [31]:
# finding the 1st quartile
q1 = np.quantile(df['Concavity'], 0.25)

# finding the 3rd quartile
q3 = np.quantile(df['Concavity'], 0.75)
med = np.median(df['Concavity'])

# finding the interquartile region (ipr)
iqr = q3-q1

# finding upper and lower whiskers
upper_bound = q3+(1.5*iqr)
lower_bound = q1-(1.5*iqr)
print('iqr:         ', iqr, '\nupper_bound: ', upper_bound, '\nlower_bound: ', lower_bound)
```

```
iqr:          0.1215475
upper_bound:  0.35092124999999996
lower_bound:  -0.13526875
```

Figure 19: Finding the Upper & Lower Bounds of "Concavity"

Now that we have calculated the upper bound and the lower bound of “Concavity” column, we can use it to find the exact outliers, and their values. Figure 20 below shows how it is done.

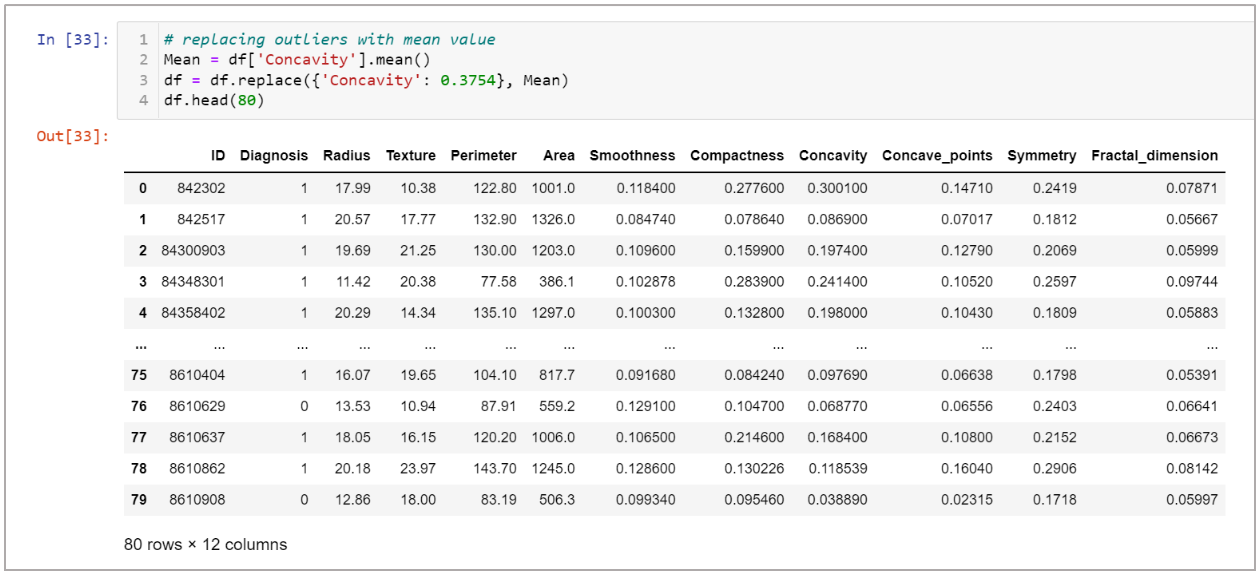


```
In [32]:
outliers = df['Concavity'][(df['Concavity'] <= lower_bound) | (df['Concavity'] >= upper_bound)]
print('Outlier Count: ', outliers.count())
print('The following are the outliers in the boxplot:\n {} '.format(outliers))
```

```
Outlier Count:  1
The following are the outliers in the boxplot:
 78    0.3754
Name: Concavity, dtype: float64
```

Figure 20: Finding Outlier using the Bounds of "Concavity"

Now that we found the exact outlier, we can replace it with the mean value of the “Concavity” column. It is shown below on Figure 21.



```
In [33]:
# replacing outliers with mean value
Mean = df['Concavity'].mean()
df = df.replace({'Concavity': 0.3754}, Mean)
df.head(80)
```

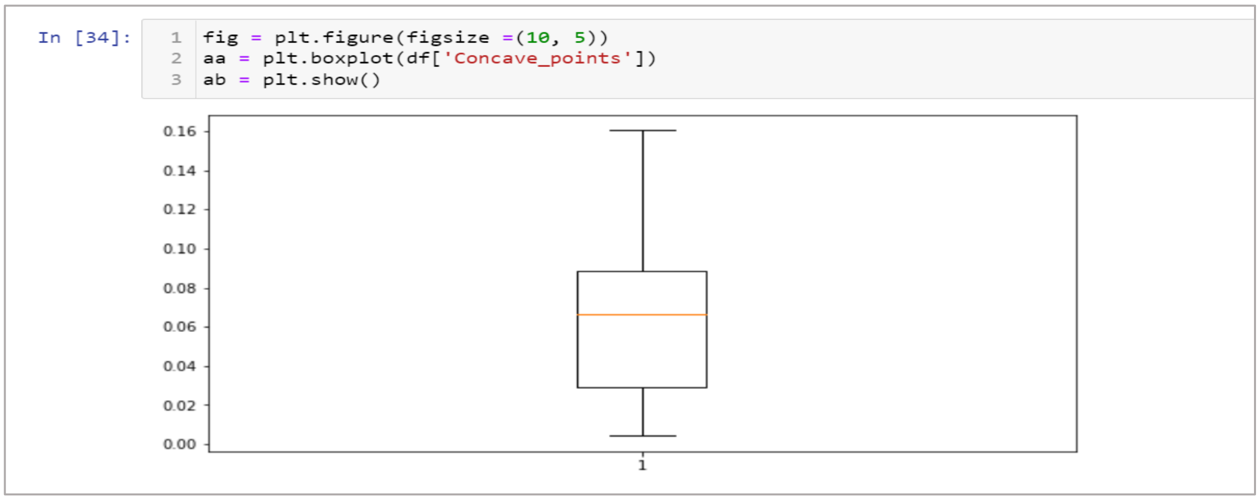
Out[33]:

	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	1	17.99	10.38	122.80	1001.0	0.118400	0.277600	0.300100	0.14710	0.2419	0.07871
1	842517	1	20.57	17.77	132.90	1326.0	0.084740	0.078640	0.086900	0.07017	0.1812	0.05667
2	84300903	1	19.69	21.25	130.00	1203.0	0.109600	0.159900	0.197400	0.12790	0.2069	0.05999
3	84348301	1	11.42	20.38	77.58	386.1	0.102878	0.283900	0.241400	0.10520	0.2597	0.09744
4	84358402	1	20.29	14.34	135.10	1297.0	0.100300	0.132800	0.198000	0.10430	0.1809	0.05883
...	...	...	...	...	...	...	...	...	...	...	...	...
75	8610404	1	16.07	19.65	104.10	817.7	0.091680	0.084240	0.097690	0.06638	0.1798	0.05391
76	8610629	0	13.53	10.94	87.91	559.2	0.129100	0.104700	0.068770	0.06556	0.2403	0.06641
77	8610637	1	18.05	16.15	120.20	1006.0	0.106500	0.214600	0.168400	0.10800	0.2152	0.06673
78	8610862	1	20.18	23.97	143.70	1245.0	0.128600	0.130226	0.118539	0.16040	0.2906	0.08142
79	8610908	0	12.86	18.00	83.19	506.3	0.099340	0.095460	0.038890	0.02315	0.1718	0.05997

80 rows × 12 columns

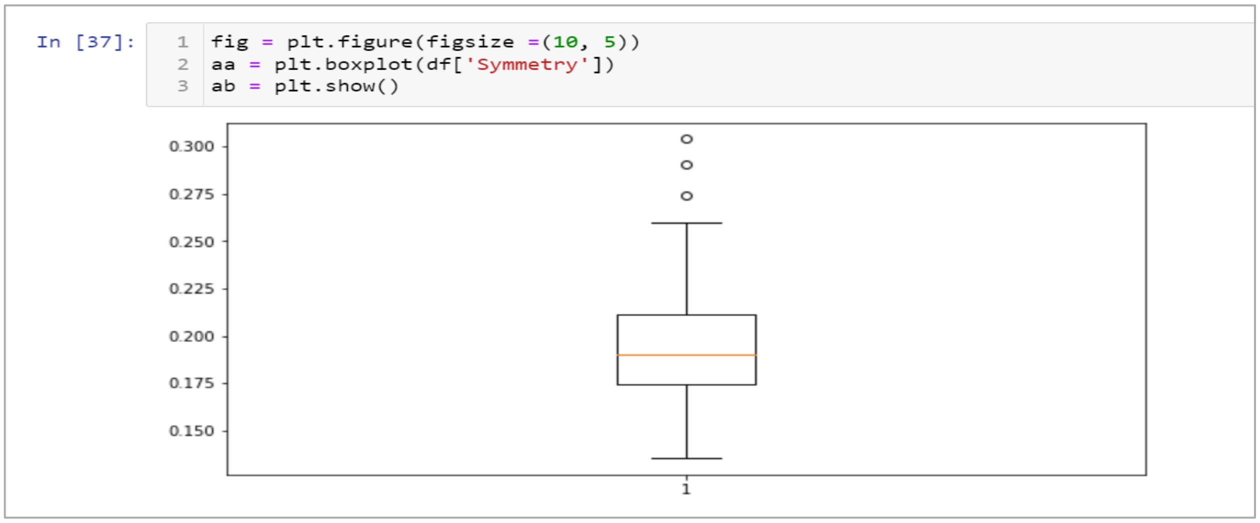
Figure 21: Replacing Outlier of "Concavity" with Mean Value

Again, now we continue finding outliers of rest of the columns.



```
In [34]:  fig = plt.figure(figsize =(10, 5))
          aa = plt.boxplot(df['Concave_points'])
          ab = plt.show()
```

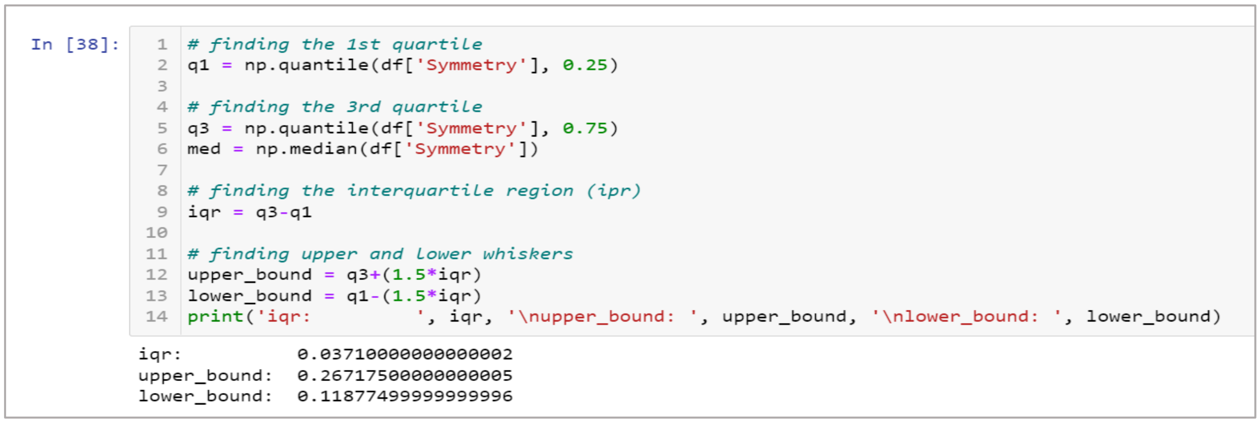
Figure 22: Boxplot of the "Concave_points" Column



```
In [37]:  fig = plt.figure(figsize =(10, 5))
          aa = plt.boxplot(df['Symmetry'])
          ab = plt.show()
```

Figure 23: Boxplot of the "Symmetry" Column

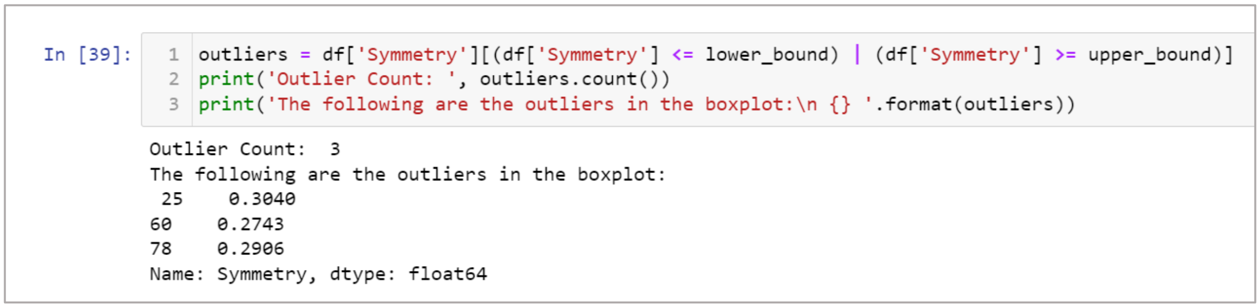
Figure 23 shows that the “Symmetry” column has a few outliers. Now we will find the bounds of this column, as shown in Figure 24.



```
In [38]:  # finding the 1st quartile
          q1 = np.quantile(df['Symmetry'], 0.25)

          # finding the 3rd quartile
          q3 = np.quantile(df['Symmetry'], 0.75)
          med = np.median(df['Symmetry'])

          # finding the interquartile region (ipr)
          iqr = q3-q1

          # finding upper and lower whiskers
          upper_bound = q3+(1.5*iqr)
          lower_bound = q1-(1.5*iqr)
          print('iqr:          ', iqr, '\nupper_bound: ', upper_bound, '\nlower_bound: ', lower_bound)

iqr:           0.03710000000000002
upper_bound:   0.26717500000000005
lower_bound:   0.11877499999999996
```

Figure 24: Finding the Upper & Lower Bounds of " Symmetry"

Now that we have calculated the upper bound and the lower bound of “Symmetry” column, we can use it to find the exact outliers, and their values. Figure 25 below shows how it is done.

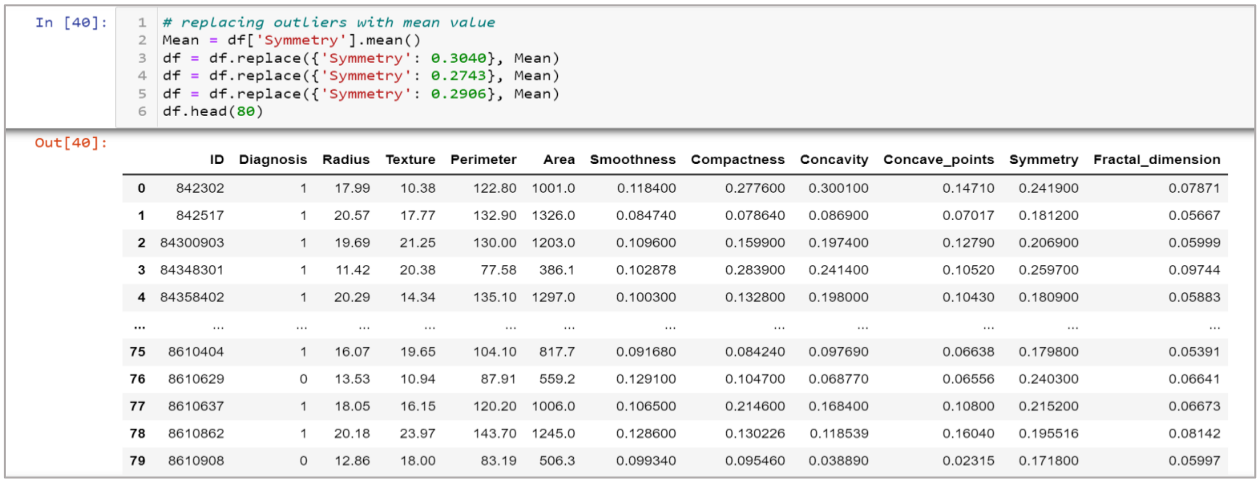


```
In [39]:
outliers = df['Symmetry'][(df['Symmetry'] <= lower_bound) | (df['Symmetry'] >= upper_bound)]
print('Outlier Count: ', outliers.count())
print('The following are the outliers in the boxplot:\n {} '.format(outliers))
```

```
Outlier Count:  3
The following are the outliers in the boxplot:
 25    0.3040
60    0.2743
78    0.2906
Name: Symmetry, dtype: float64
```

Figure 25: Finding Outlier using the Bounds of "Symmetry"

Now that we found the exact outliers, we can replace it with the mean value of the “Symmetry” column. It is shown below on Figure 26.



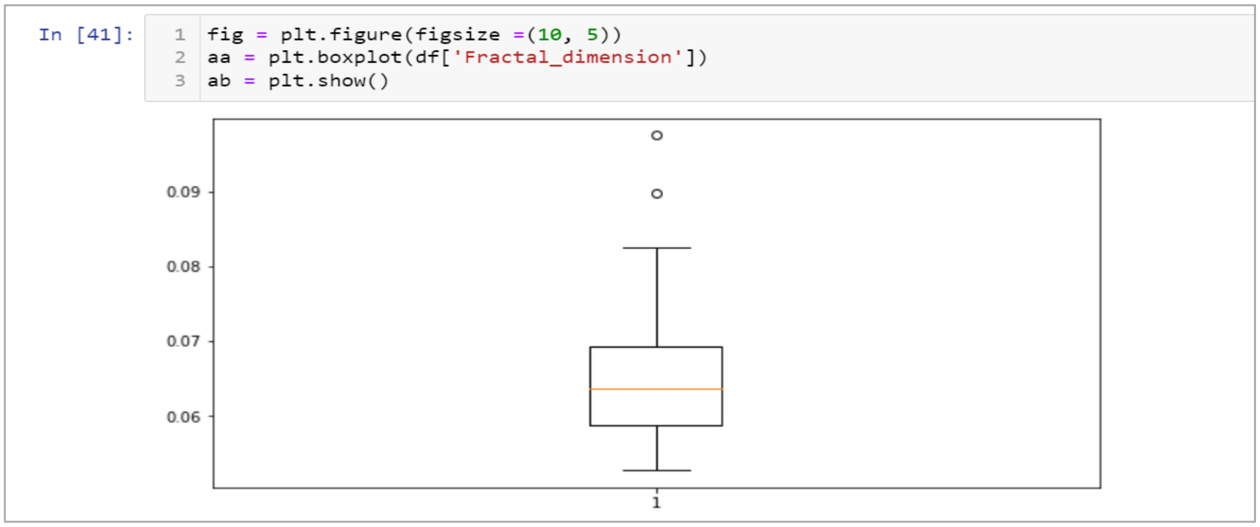
```
In [40]:
# replacing outliers with mean value
Mean = df['Symmetry'].mean()
df = df.replace({'Symmetry': 0.3040}, Mean)
df = df.replace({'Symmetry': 0.2743}, Mean)
df = df.replace({'Symmetry': 0.2906}, Mean)
df.head(80)
```

Out[40]:

	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	1	17.99	10.38	122.80	1001.0	0.118400	0.277600	0.300100	0.14710	0.241900	0.07871
1	842517	1	20.57	17.77	132.90	1326.0	0.084740	0.078640	0.086900	0.07017	0.181200	0.05667
2	84300903	1	19.69	21.25	130.00	1203.0	0.109600	0.159900	0.197400	0.12790	0.206900	0.05999
3	84348301	1	11.42	20.38	77.58	386.1	0.102878	0.283900	0.241400	0.10520	0.259700	0.09744
4	84358402	1	20.29	14.34	135.10	1297.0	0.100300	0.132800	0.198000	0.10430	0.180900	0.05883
...	...	...	...	...	...	...	...	...	...	...	...	...
75	8610404	1	16.07	19.65	104.10	817.7	0.091680	0.084240	0.097690	0.06638	0.179800	0.05391
76	8610629	0	13.53	10.94	87.91	559.2	0.129100	0.104700	0.068770	0.06556	0.240300	0.06641
77	8610637	1	18.05	16.15	120.20	1006.0	0.106500	0.214600	0.168400	0.10800	0.215200	0.06673
78	8610862	1	20.18	23.97	143.70	1245.0	0.128600	0.130226	0.118539	0.16040	0.195516	0.08142
79	8610908	0	12.86	18.00	83.19	506.3	0.099340	0.095460	0.038890	0.02315	0.171800	0.05997

Figure 26: Replacing Outlier of "Symmetry" with Mean Value

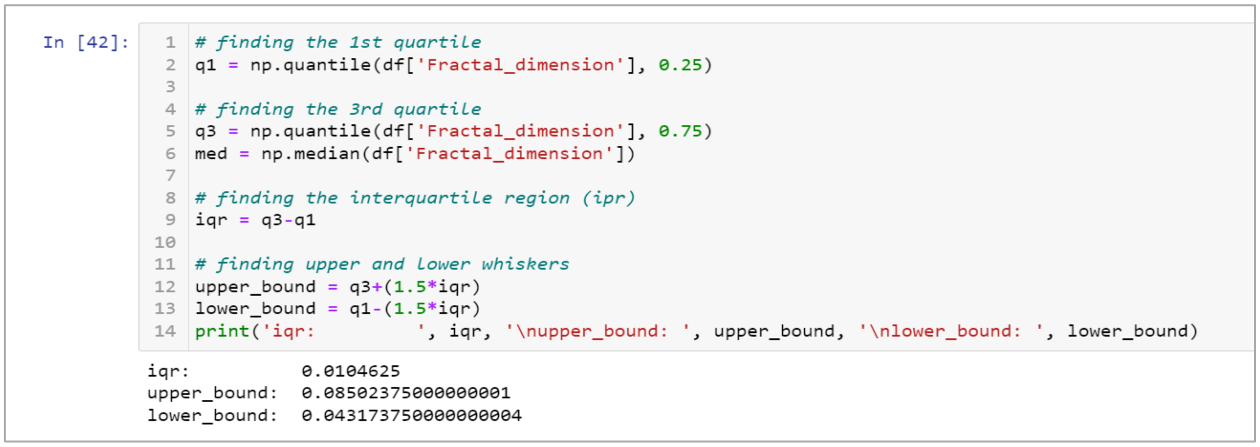
Again, now we continue finding outliers of rest of the columns.



```
In [41]:
fig = plt.figure(figsize =(10, 5))
aa = plt.boxplot(df['Fractal_dimension'])
ab = plt.show()
```

Figure 27: Boxplot of the "Fractal_dimension" Column

Figure 27 shows that the "Fractal_dimension" column has a couple of outliers. Now we will find the bounds of this column, as shown in Figure 28.

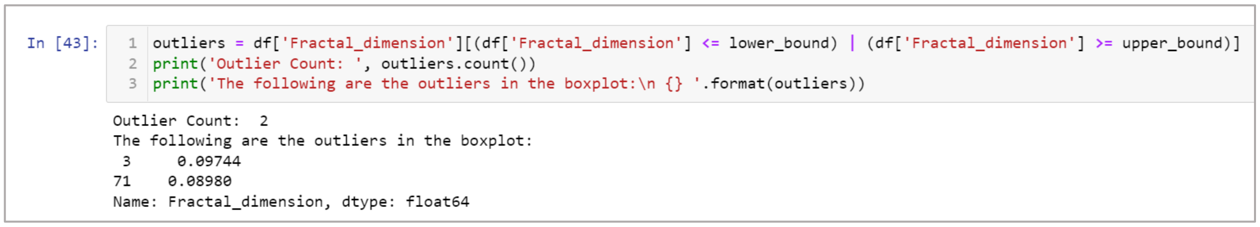


```
In [42]:
# finding the 1st quartile
q1 = np.quantile(df['Fractal_dimension'], 0.25)

# finding the 3rd quartile
q3 = np.quantile(df['Fractal_dimension'], 0.75)
med = np.median(df['Fractal_dimension'])

# finding the interquartile region (ipr)
iqr = q3-q1

# finding upper and lower whiskers
upper_bound = q3+(1.5*iqr)
lower_bound = q1-(1.5*iqr)
print('iqr:          ', iqr, '\nupper_bound: ', upper_bound, '\nlower_bound: ', lower_bound)
```

```
iqr:          0.0104625
upper_bound:  0.08502375000000001
lower_bound:  0.043173750000000004
```

Figure 28: Finding the Upper & Lower Bounds of " Fractal_dimension"

Now that we have calculated the upper and the lower bounds of "Fractal_dimension" column, we can use it to find the exact outliers, and their values. Figure 29 below shows how it is done.

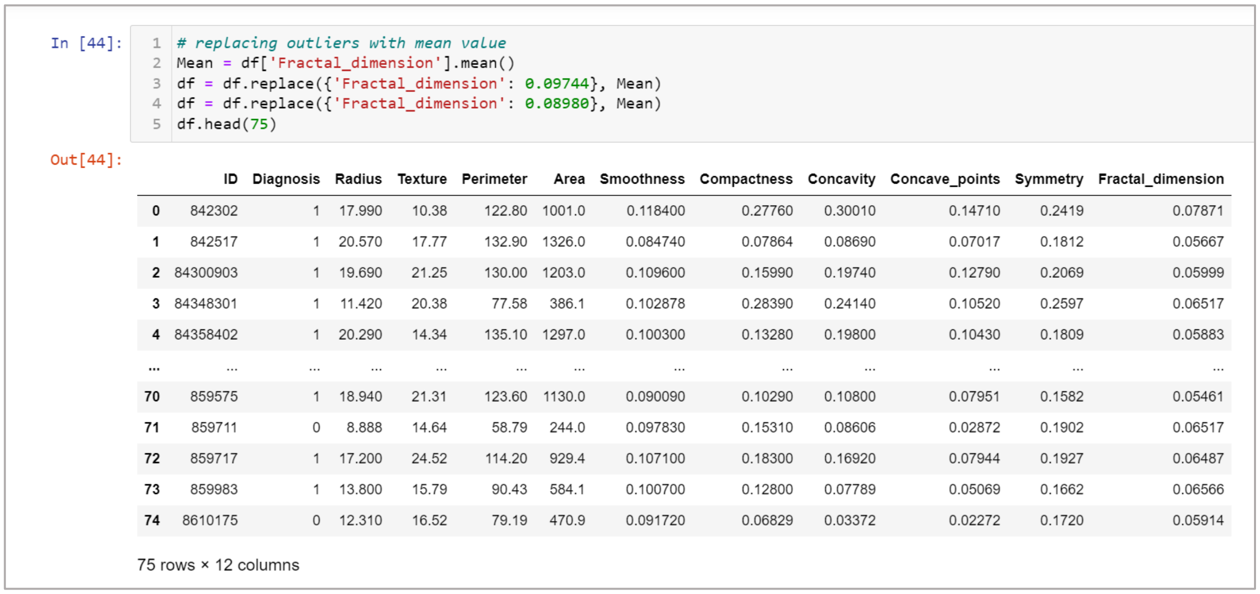


```
In [43]:
outliers = df['Fractal_dimension'][(df['Fractal_dimension'] <= lower_bound) | (df['Fractal_dimension'] >= upper_bound)]
print('Outlier Count: ', outliers.count())
print('The following are the outliers in the boxplot:\n {} '.format(outliers))
```

```
Outlier Count:  2
The following are the outliers in the boxplot:
 3     0.09744
71    0.08980
Name: Fractal_dimension, dtype: float64
```

Figure 29: Finding Outlier using the Bounds of "Fractal_dimension"

Now that we found the exact outliers, we can replace them with the mean value of the "Fractal_ dimension" column. It is shown below on Figure 30.



```
In [44]:
# replacing outliers with mean value
Mean = df['Fractal_dimension'].mean()
df = df.replace({'Fractal_dimension': 0.09744}, Mean)
df = df.replace({'Fractal_dimension': 0.08980}, Mean)
df.head(75)
```

Out[44]:

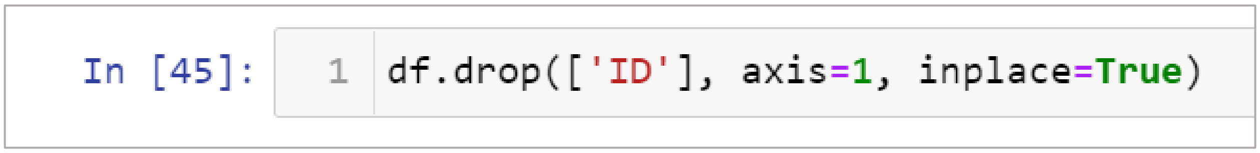
	ID	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
0	842302	1	17.990	10.38	122.80	1001.0	0.118400	0.27760	0.30010	0.14710	0.2419	0.07871
1	842517	1	20.570	17.77	132.90	1326.0	0.084740	0.07864	0.08690	0.07017	0.1812	0.05667
2	84300903	1	19.690	21.25	130.00	1203.0	0.109600	0.15990	0.19740	0.12790	0.2069	0.05999
3	84348301	1	11.420	20.38	77.58	386.1	0.102878	0.28390	0.24140	0.10520	0.2597	0.06517
4	84358402	1	20.290	14.34	135.10	1297.0	0.100300	0.13280	0.19800	0.10430	0.1809	0.05883
...	...	...	...	...	...	...	...	...	...	...	...	...
70	859575	1	18.940	21.31	123.60	1130.0	0.090090	0.10290	0.10800	0.07951	0.1582	0.05461
71	859711	0	8.888	14.64	58.79	244.0	0.097830	0.15310	0.08606	0.02872	0.1902	0.06517
72	859717	1	17.200	24.52	114.20	929.4	0.107100	0.18300	0.16920	0.07944	0.1927	0.06487
73	859983	1	13.800	15.79	90.43	584.1	0.100700	0.12800	0.07789	0.05069	0.1662	0.06566
74	8610175	0	12.310	16.52	79.19	470.9	0.091720	0.06829	0.03372	0.02272	0.1720	0.05914

75 rows × 12 columns

Figure 30: Replacing Outlier of "Fractal_dimension" with Mean Value

2.5 Dropping Irrelevant Data Columns

Now since we know that the "**ID**" column has no relation with, or contribution to, the calculation of the Diagnosis, so hence we will drop the "ID" column, since it is irrelevant to the dataset and to the creation of the machine learning model.

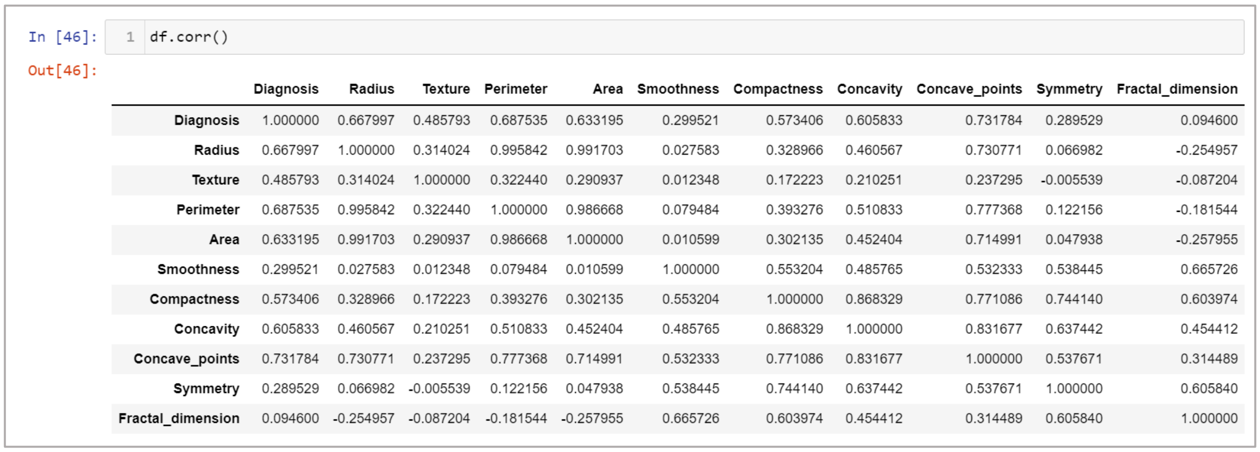


```
In [45]:  df.drop(['ID'], axis=1, inplace=True)
```

Figure 31: Dropping the "ID" column

2.6 Data Correlation

Finally, now that we have cleaned maximum amount of the dirty and noisy data from the dataset, we can proceed with the next stage, which is to find the correlation of the columns in relation to each other. Figure 32 shows the correlation table of the dataset.

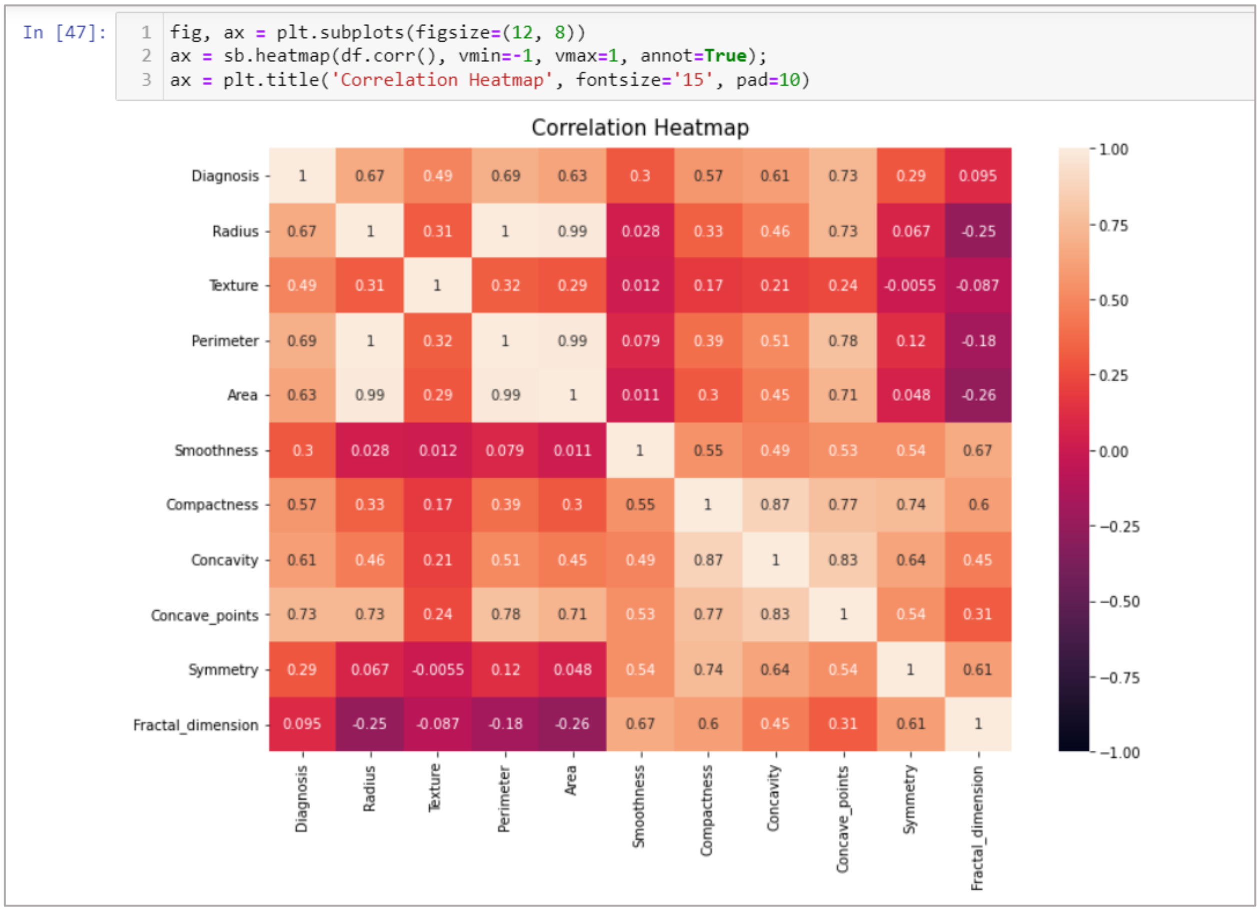


```
In [46]:  df.corr()
Out[46]:
```

	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
Diagnosis	1.000000	0.667997	0.485793	0.687535	0.633195	0.299521	0.573406	0.605833	0.731784	0.289529	0.094600
Radius	0.667997	1.000000	0.314024	0.995842	0.991703	0.027583	0.328966	0.460567	0.730771	0.066982	-0.254957
Texture	0.485793	0.314024	1.000000	0.322440	0.290937	0.012348	0.172223	0.210251	0.237295	-0.005539	-0.087204
Perimeter	0.687535	0.995842	0.322440	1.000000	0.986668	0.079484	0.393276	0.510833	0.777368	0.122156	-0.181544
Area	0.633195	0.991703	0.290937	0.986668	1.000000	0.010599	0.302135	0.452404	0.714991	0.047938	-0.257955
Smoothness	0.299521	0.027583	0.012348	0.079484	0.010599	1.000000	0.553204	0.485765	0.532333	0.538445	0.665726
Compactness	0.573406	0.328966	0.172223	0.393276	0.302135	0.553204	1.000000	0.868329	0.771086	0.744140	0.603974
Concavity	0.605833	0.460567	0.210251	0.510833	0.452404	0.485765	0.868329	1.000000	0.831677	0.637442	0.454412
Concave_points	0.731784	0.730771	0.237295	0.777368	0.714991	0.532333	0.771086	0.831677	1.000000	0.537671	0.314489
Symmetry	0.289529	0.066982	-0.005539	0.122156	0.047938	0.538445	0.744140	0.637442	0.537671	1.000000	0.605840
Fractal_dimension	0.094600	-0.254957	-0.087204	-0.181544	-0.257955	0.665726	0.603974	0.454412	0.314489	0.605840	1.000000

Figure 32: Correlation Table of the Dataset

Based on this correlation, we will make the correlation heatmap of the dataset. This will be to help ease the process of finding the relation of columns with each other. The image below (Figure 33) illustrates the correlation heatmap of the dataset, after it is cleaned.



```
In [47]:
fig, ax = plt.subplots(figsize=(12, 8))
ax = sb.heatmap(df.corr(), vmin=-1, vmax=1, annot=True);
ax = plt.title('Correlation Heatmap', fontsize='15', pad=10)
```

Correlation Heatmap

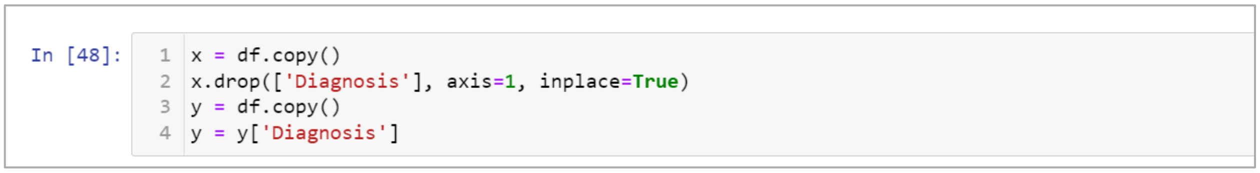
	Diagnosis	Radius	Texture	Perimeter	Area	Smoothness	Compactness	Concavity	Concave_points	Symmetry	Fractal_dimension
Diagnosis	1	0.67	0.49	0.69	0.63	0.3	0.57	0.61	0.73	0.29	0.095
Radius	0.67	1	0.31	1	0.99	0.028	0.33	0.46	0.73	0.067	-0.25
Texture	0.49	0.31	1	0.32	0.29	0.012	0.17	0.21	0.24	-0.0055	-0.087
Perimeter	0.69	1	0.32	1	0.99	0.079	0.39	0.51	0.78	0.12	-0.18
Area	0.63	0.99	0.29	0.99	1	0.011	0.3	0.45	0.71	0.048	-0.26
Smoothness	0.3	0.028	0.012	0.079	0.011	1	0.55	0.49	0.53	0.54	0.67
Compactness	0.57	0.33	0.17	0.39	0.3	0.55	1	0.87	0.77	0.74	0.6
Concavity	0.61	0.46	0.21	0.51	0.45	0.49	0.87	1	0.83	0.64	0.45
Concave_points	0.73	0.73	0.24	0.78	0.71	0.53	0.77	0.83	1	0.54	0.31
Symmetry	0.29	0.067	-0.0055	0.12	0.048	0.54	0.74	0.64	0.54	1	0.61
Fractal_dimension	0.095	-0.25	-0.087	-0.18	-0.26	0.67	0.6	0.45	0.31	0.61	1

Figure 33: Correlation Heatmap of the Cleaned Dataset

3.0 Developing Prediction Model

3.1 Setting Independent & Dependent Variables

Now, we will make the dependent and independent variables. Since all the other columns, will be used to calculate the Diagnosis, they all shall be the input data, and thus the independent variables. And as the Diagnosis is the output that all columns calculate, the Diagnosis will be the dependent variable. So, we will set all the independent variables as **x** and the diagnosis as **y**, as shown below. x is set by first making a copy of the cleaned dataset, then removing the diagnosis columns, and the y is set by making copy of the dataset, and only selecting the Diagnosis.

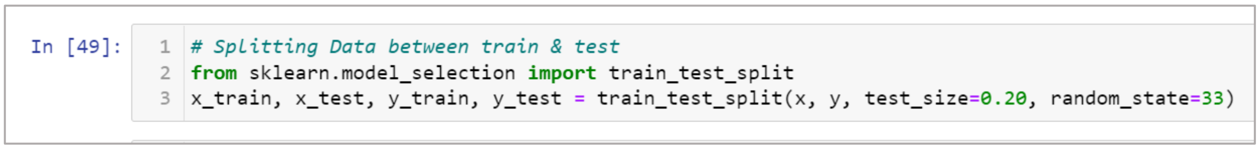


```
In [48]:
x = df.copy()
x.drop(['Diagnosis'], axis=1, inplace=True)
y = df.copy()
y = y['Diagnosis']
```

Figure 34: Setting x (Inputs) & y (Output)

3.2 Splitting Data Into Training & Testing Parts

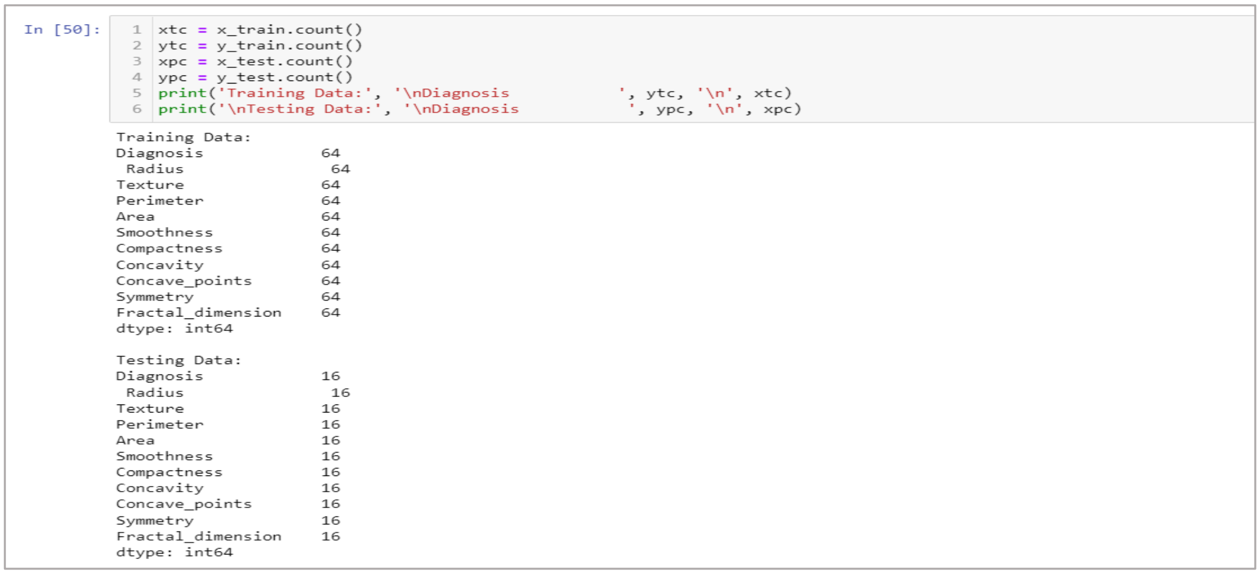
Before we start making the actual model, we will need to split the dataset into two parts, training and testing. We can move a portion of the dataset into a new one, but that process is a manual and inefficient way of dividing the data, since we will have to manually make a new csv file, and manually move a number of rows from the main dataset to the new one. Instead, we can use a more efficient way of splitting the dataset into training and testing parts, which is to use the "train_test_split" method that is included in the Scikit Learn module. We can use the "train_test _split" method to automatically split the data into the two intended parts by just calling the function, and giving the ratio of the test size, which would be for testing, and the rest of the dataset will be for training. We can also mention the level of randomness of selecting the data for splitting. The image below (Figure 35) shows how this module is being used to split the data.



```
In [49]:
# Splitting Data between train & test
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=33)
```

Figure 35: Splitting Data Using Scikit Learn into Training & Testing Parts

As we can see from the image above, the "x_train" and the "y_train" are the input and output values for the training, whereas the "x_test" and "y_test" are for testing the input against the output values of the model. Note, the "test_size" is mentioned to be **0.2**, meaning 20% of the data will be used for the testing phase, thus the rest 80% will be used for the training phase.



```
In [50]:
xtc = x_train.count()
ytc = y_train.count()
xpc = x_test.count()
ypc = y_test.count()
print('Training Data:', '\nDiagnosis            ', ytc, '\n', xtc)
print('\nTesting Data:', '\nDiagnosis            ', ypc, '\n', xpc)
```

```
Training Data:
Diagnosis            64
 Radius               64
Texture              64
Perimeter            64
Area                 64
Smoothness           64
Compactness          64
Concavity            64
Concave_points       64
Symmetry             64
Fractal_dimension    64
dtype: int64

Testing Data:
Diagnosis            16
 Radius               16
Texture              16
Perimeter            16
Area                 16
Smoothness           16
Compactness          16
Concavity            16
Concave_points       16
Symmetry             16
Fractal_dimension    16
dtype: int64
```

Figure 36: Showing the Training & Testing Data Count

As we can see from Figure 36, the training set has 64 data rows, and the testing set has 16 rows.

3.3 Scaling data

Since we will be using Neural Network modelling, the deep learning algorithm will multiply the input data and its weight, which will take significant time to process and make the model. As a result, we will need to scale the data to reduce the time and processing complexity, essentially optimizing the data for a better deep learning performance. The image below shows the process of scaling the data.

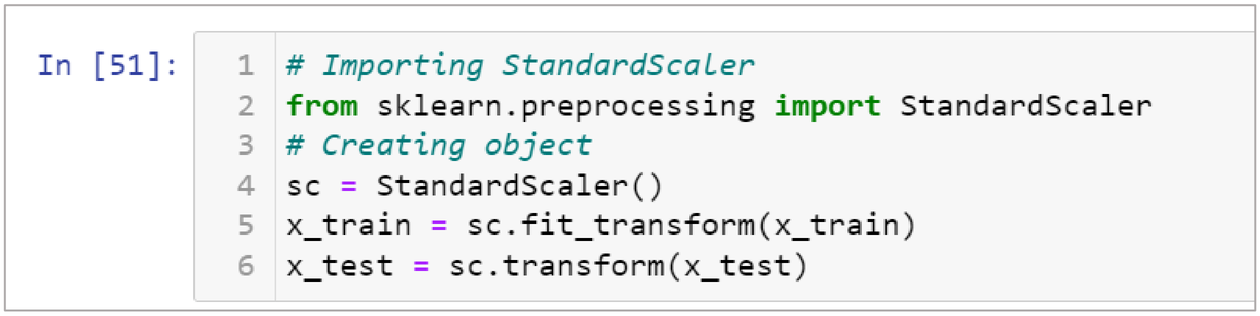

```
In [51]:
# Importing StandardScaler
from sklearn.preprocessing import StandardScaler
# Creating object
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

Figure 37: Scaling the Data Using "StandardScaler"

3.4 Making the Artificial Neural Network Model

Now that we have our data cleaned, split, scaled and everything prepared for the modelling, we are finally ready to create the actual neural model of the training. We import and use the **Keras** package. We use the **Sequential** algorithm to create the model, and add **Dense** layers to make the input layer, hidden layer, and output layer of the model. For the hidden layers we use the **relu** activation function, while for the output we use **sigmoid** as the activation function. The image below (Figure 38) illustrates the process talked in this paragraph.

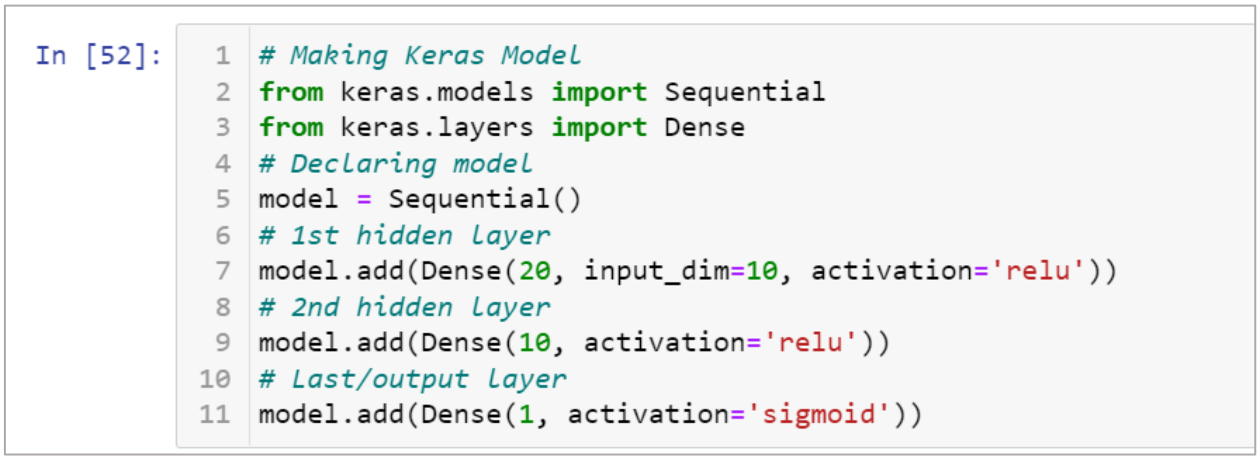
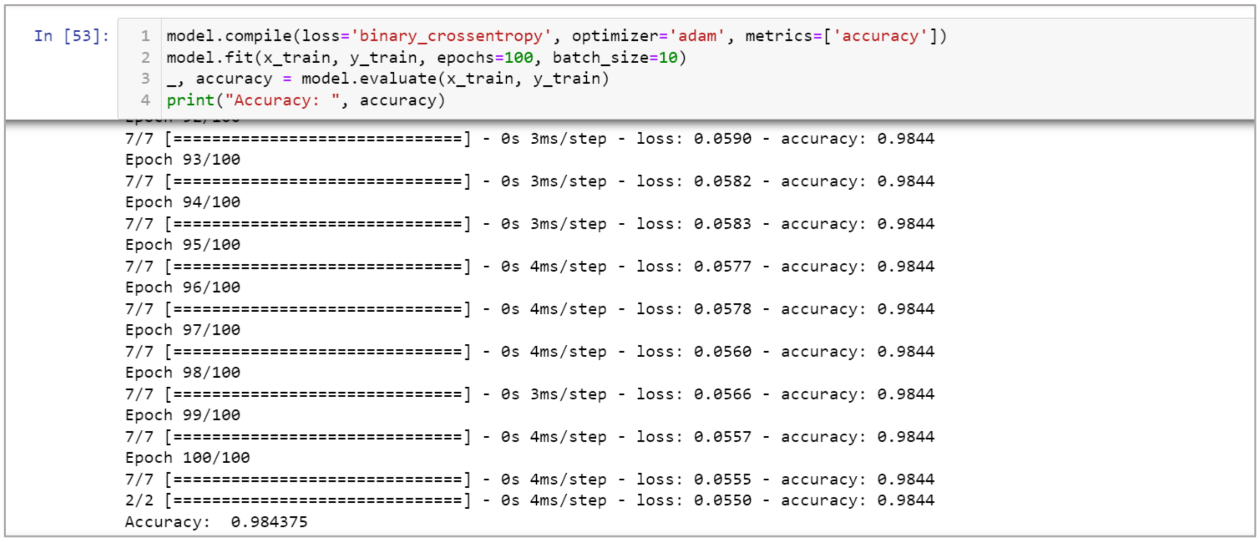

```
In [52]:
# Making Keras Model
from keras.models import Sequential
from keras.layers import Dense
# Declaring model
model = Sequential()
# 1st hidden layer
model.add(Dense(20, input_dim=10, activation='relu'))
# 2nd hidden layer
model.add(Dense(10, activation='relu'))
# Last/output layer
model.add(Dense(1, activation='sigmoid'))
```

Figure 38: Making the Artificial Neural Network Model

3.5 Training Model & Improve Accuracy

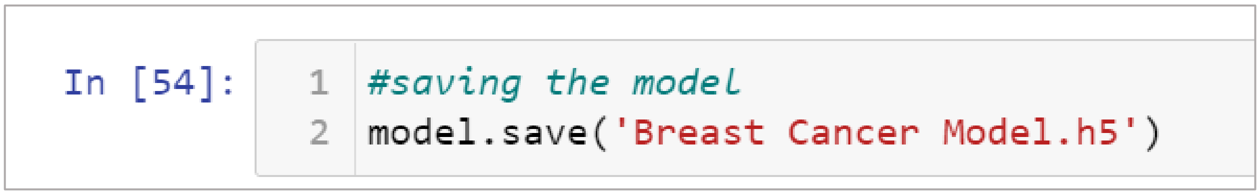
After making the model, we will now input the training input data (x_train) and check the output with the training output (y_train), and see the accuracy of the model. For our case, we gave a **epoch** (means the training cycle) of 100, and the **batch size** (which is the number of samples processed before updating model) as 10. Figure 39 below illustrates the process of training the model and showing the accuracy of training each cycle (till the last training cycle).



```
In [53]: model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
         model.fit(x_train, y_train, epochs=100, batch_size=10)
         _, accuracy = model.evaluate(x_train, y_train)
         print("Accuracy: ", accuracy)

7/7 [==============================] - 0s 3ms/step - loss: 0.0590 - accuracy: 0.9844
Epoch 93/100
7/7 [==============================] - 0s 3ms/step - loss: 0.0582 - accuracy: 0.9844
Epoch 94/100
7/7 [==============================] - 0s 3ms/step - loss: 0.0583 - accuracy: 0.9844
Epoch 95/100
7/7 [==============================] - 0s 4ms/step - loss: 0.0577 - accuracy: 0.9844
Epoch 96/100
7/7 [==============================] - 0s 4ms/step - loss: 0.0578 - accuracy: 0.9844
Epoch 97/100
7/7 [==============================] - 0s 4ms/step - loss: 0.0560 - accuracy: 0.9844
Epoch 98/100
7/7 [==============================] - 0s 3ms/step - loss: 0.0566 - accuracy: 0.9844
Epoch 99/100
7/7 [==============================] - 0s 4ms/step - loss: 0.0557 - accuracy: 0.9844
Epoch 100/100
7/7 [==============================] - 0s 4ms/step - loss: 0.0555 - accuracy: 0.9844
2/2 [==============================] - 0s 4ms/step - loss: 0.0550 - accuracy: 0.9844
Accuracy:  0.984375
```

Figure 39: Training Model & Checking Accuracy

At the end, we can see that after the 100 epochs, the accuracy of the model reached 0.984, meaning it reached its accuracy rate at 98.4%. This means the model is trained enough to be used for implementations. We can save the model to be used later on by other applications or to transfer the model, so we don't need to train the model every time before usage. The image below (Figure 40) illustrates the process of saving the model into a "**.h5**" file.



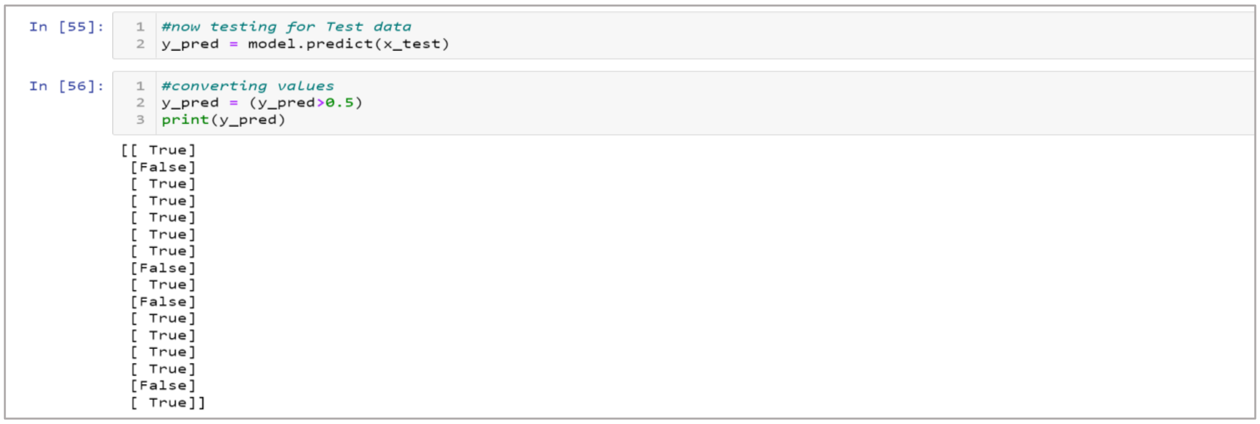
```
In [54]: #saving the model
         model.save('Breast Cancer Model.h5')
```

Figure 40: Saving the Artificial Neural Network Model

4.0 Applying Prediction Model to New Data

4.1 Testing Model Using Test Data

Now we will use the trained model to predict the outputs using the test set of the dataset (x_test and y_test). We will first predict the output of the model using the input data of the test set (x_test). Then we will convert the output into a **Boolean** (1, 0) format, since the output of the test set is also a binary (1 or 0). We shall then display the converted outputs. Figure 41 below shows the process of making the predictions using the model, and then converting the predictions into Booleans, and displaying them.

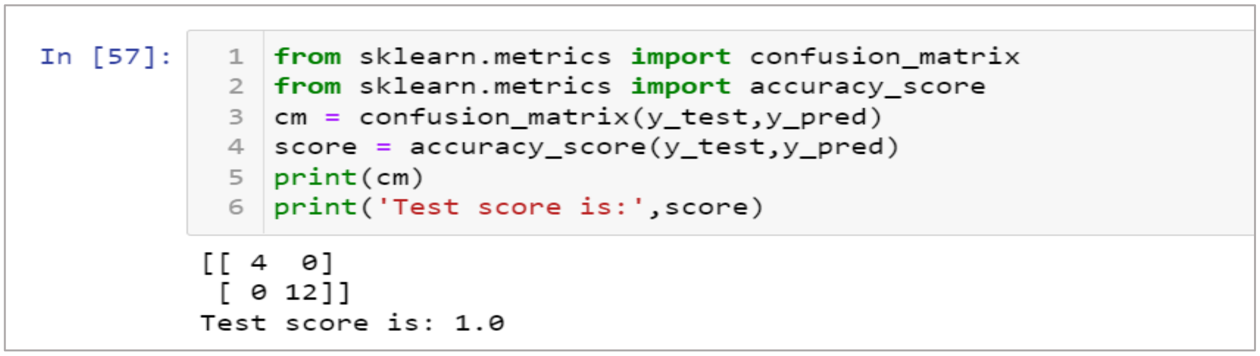


```
In [55]:  #now testing for Test data
          y_pred = model.predict(x_test)

In [56]:  #converting values
          y_pred = (y_pred>0.5)
          print(y_pred)

[[ True]
 [False]
 [ True]
 [ True]
 [ True]
 [ True]
 [ True]
 [False]
 [ True]
 [False]
 [ True]
 [ True]
 [ True]
 [ True]
 [False]
 [ True]]
```

Figure 41: Predicting Output of Test Data & Converting them to Boolean Format.

4.2 Seeing Model Score & Confusion Matrix

Now that we have converted the model prediction into binary outputs, we can compare them against the test output data (y_test) to see the model score. Aside from that we can also see the model accuracy score. Figure 42 below shows how the predicted and converted output (y_pred) is compared to the y_test to get the model's accuracy score and the confusion matrix.

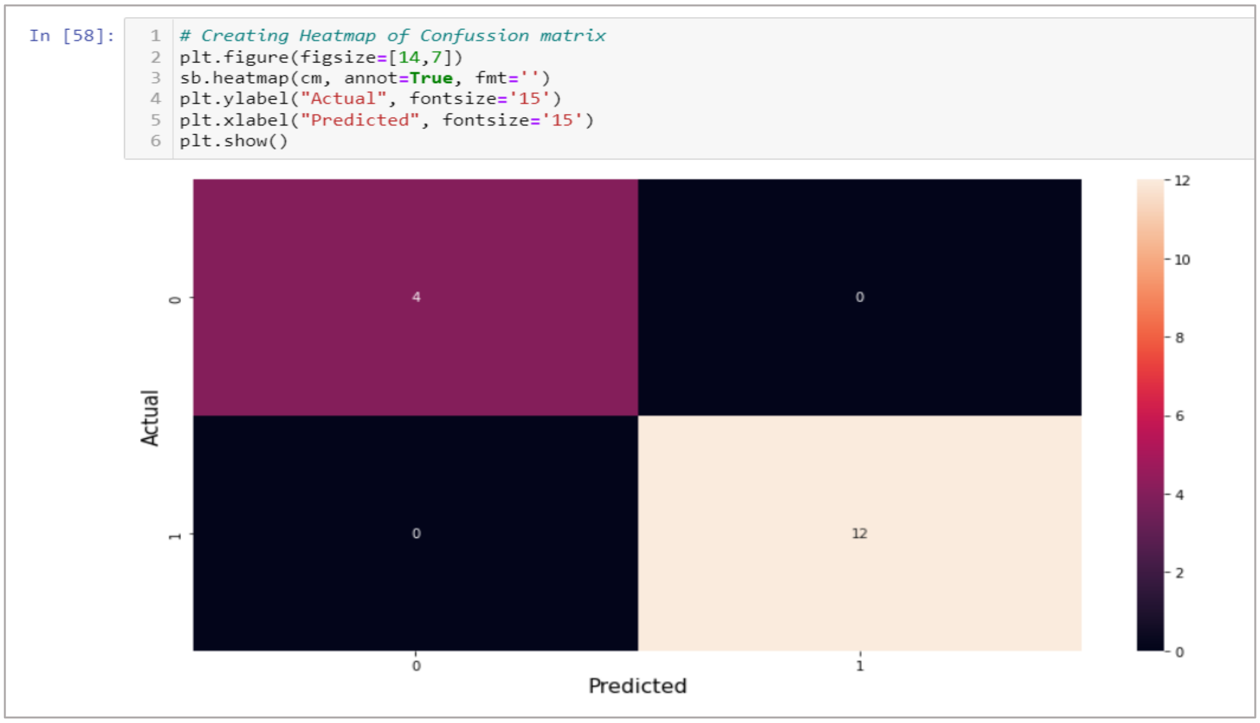


```
In [57]:  from sklearn.metrics import confusion_matrix
          from sklearn.metrics import accuracy_score
          cm = confusion_matrix(y_test,y_pred)
          score = accuracy_score(y_test,y_pred)
          print(cm)
          print('Test score is:',score)

[[ 4  0]
 [ 0 12]]
Test score is: 1.0
```

Figure 43: Displaying Confusion Matrix & Model Score

As we can see from the Figure 43, the model score (Test Score) is 1, meaning it is 100% accurate. And the confusion matrix shows that there are four **True-Negative** predictions, and 12 **True-Positive** predictions (total of 16 test predictions). This means that the model is perfect, and well trained to be implemented in real life diagnosis of future patients.

To get a better visual of the confusion matrix, we can plot the confusion matrix as a **seaborn** heatmap. The Figure 44 below illustrates the process of creating the heatmap of the confusion matrix, to get a better and more understandable view of the matrix.



```
In [58]:  # Creating Heatmap of Confussion matrix
          plt.figure(figsize=[14,7])
          sb.heatmap(cm, annot=True, fmt='')
          plt.ylabel("Actual", fontsize='15')
          plt.xlabel("Predicted", fontsize='15')
          plt.show()
```

Figure 44: Confusion Matrix Heatmap

5.0 Conclusion

So, we have taken a dataset with many (more than 5) columns and 80 rows (more than 20 items). We used Numpy and Pandas to format the data and do proper data cleaning to take out missing values, clean outliers, omit out irrelevant columns, convert data to its proper data type (converting qualitative data to quantitative data). Later we split the data into training and testing parts, and scaled each set of data to optimize it for Deep Learning. Soon after, we trained the Artificial Neural Network model with the data till it had a significantly high accuracy. Lastly, we tested the model with the test data, and found a perfect accuracy score and confusion matrix. This shows that our modelling process was executed perfectly and was a success.