# DSECL ZG565 - MACINE LEARNING - ASSIGNMENT I

## Group 043


### Assignment Done By:

1. RITIKA ARORA (2021FC04617)
2. HARIT KUMAR (2021FC04184)
3. SYED SAQLAIN AHMED (2021FC04177)


## Aim of the Project

We have recieved data for seven different types of dry beans. In order to develop a supervised macine lerning model, we need to take into account different features of these beans such as form, shape, type, and structure by the market situation which will in turn help us distinguish different varieties of beans that have high feature similarity. The dataset provided consists of data for 13,611 grains of 7 different types which have and a total of 16 features; 12 dimensions and 4 shape forms.

The aim of this project is to develop a supervised machine learning algorithm to perform a multi-classification of dry beans species harvested from population cultivation from a single farm.

## Importing Libraries

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler, RobustScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.decomposition import PCA

from sklearn.linear_model  import Ridge,Lasso,RidgeCV, LassoCV, ElasticNet, ElasticNetCV, LogisticRegression, SGDClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve, roc_auc_score, classification_report
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, classification_report, confusion_matrix

from sklearn.model_selection import cross_val_score,KFold,StratifiedKFold,LeaveOneOut

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

## Importing Dataset

In [2]:
```
df_root = pd.read_excel("Dry_Bean_Dataset.xlsx")
df_root.head(2)
```

Out[2]:

| | Bean ID | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent | Solidity | roundness |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1.0 | 28395.0 | 610.291 | 208.178117 | 173.888747 | 1.197191 | 0.549812 | 28715.0 | 190.141097 | 0.763923 | 0.988856 | 0.958027 |
| **1** | 2.0 | 28734.0 | 638.018 | 200.524796 | 182.734419 | 1.097356 | 0.411785 | 29172.0 | 191.272750 | 0.783968 | 0.984986 | 0.887034 |

## Exploratory Data Analysis

Exploratory Data Analysis (EDA) is an approach to analyze the data using visual techniques. It is used to discover trends, patterns, or to check assumptions with the help of statistical summary and graphical representations.

## Initial

1.At the independent columns are continuous and target(dependent column) is a categorical variable

1. Total columns in the dataset are 18 where 17 are Independent features and 1 is a dependent features

In [3]:
```
df_root.describe()
```

Out[3]:

| | Bean ID | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | |
|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 13611.000000 | 13611.000000 | 13611.000000 | 13609.000000 | 13608.000000 | 13610.000000 | 13609.000000 | 13611.000000 | 13611.000000 | 1361 |
| **mean** | 6806.000000 | 53048.284549 | 855.283459 | 320.155372 | 202.273996 | 1.583268 | 0.750930 | 53768.200206 | 253.064220 | |
| **std** | 3929.301592 | 29324.095717 | 214.289696 | 85.693199 | 44.974478 | 0.246669 | 0.091962 | 29774.915817 | 59.177120 | |
| **min** | 1.000000 | 20420.000000 | 524.736000 | 183.601165 | 122.512653 | 1.024868 | 0.218951 | 20684.000000 | 161.243764 | |
| **25%** | 3403.500000 | 36328.000000 | 703.523500 | 253.319280 | 175.840519 | 1.432352 | 0.715953 | 36714.500000 | 215.068003 | |
| **50%** | 6806.000000 | 44652.000000 | 794.941000 | 296.899313 | 192.443880 | 1.551132 | 0.764446 | 45178.000000 | 238.438026 | |
| **75%** | 10208.500000 | 61332.000000 | 977.213000 | 376.497678 | 217.036082 | 1.707118 | 0.810471 | 62294.000000 | 279.446467 | |
| **max** | 13611.000000 | 254616.000000 | 1985.370000 | 738.860153 | 460.198497 | 2.430306 | 0.911423 | 263261.000000 | 569.374358 | |

In [4]:
```
df_root.dtypes
```

Out[4]:
```
Bean ID            float64
Area               float64
Perimeter          float64
MajorAxisLength    float64
MinorAxisLength    float64
AspectRation       float64
Eccentricity       float64
ConvexArea         float64
EquivDiameter      float64
Extent             float64
Solidity           float64
roundness          float64
Compactness        float64
ShapeFactor1       float64
ShapeFactor2       float64
ShapeFactor3       float64
ShapeFactor4       float64
Class               object
dtype: object
```

**Continuous to Categorical Classification problem**

There are total of 7 Class in the target feature

In [5]:
```
df_missing = df_root.copy()
set(df_missing.Class)
```

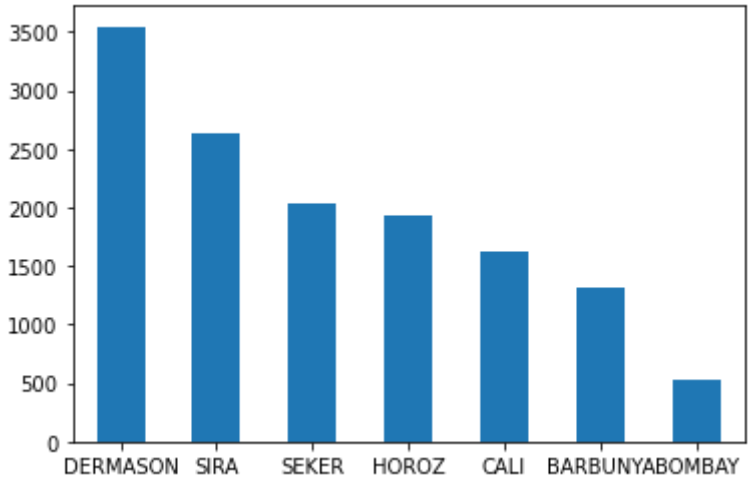
Out[5]:
```
{'BARBUNYA', 'BOMBAY', 'CALI', 'DERMASON', 'HOROZ', 'SEKER', 'SIRA'}
```

**Checking balance of the Data with respect to Target Feature**

As the number of classes in targte features are 7 we try to check if the dataset provided is a balanced or imbalanced.

In [6]:
```
count_class = pd.value_counts(df_root["Class"],sort = True)
count_class.plot(kind='bar',rot = 0)
```

Out[6]:
```
<AxesSubplot:>
```




Its is multiclass classification problem with bombay class having lesser data.

**Looking for duplicate rows**

Number of rows in the dataset that repeat is zero

In [7]:
```
df_root[df_root.duplicated()]
```

Out[7]:

| Bean ID | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent | Solidity | roundness | Com |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

## Correlational Analysis

Correlation analysis is a statistical method used to measure the strength of the linear relationship between two variables and compute their association.

Correlation analysis calculates the level of change in one variable due to the change in the other.

A high correlation points to a strong relationship between the two variables, while a low correlation means that the variables are weakly related.

In [8]:
```
df_tmp = df_root.drop(columns=['Bean ID'])
df_root.corr()
```

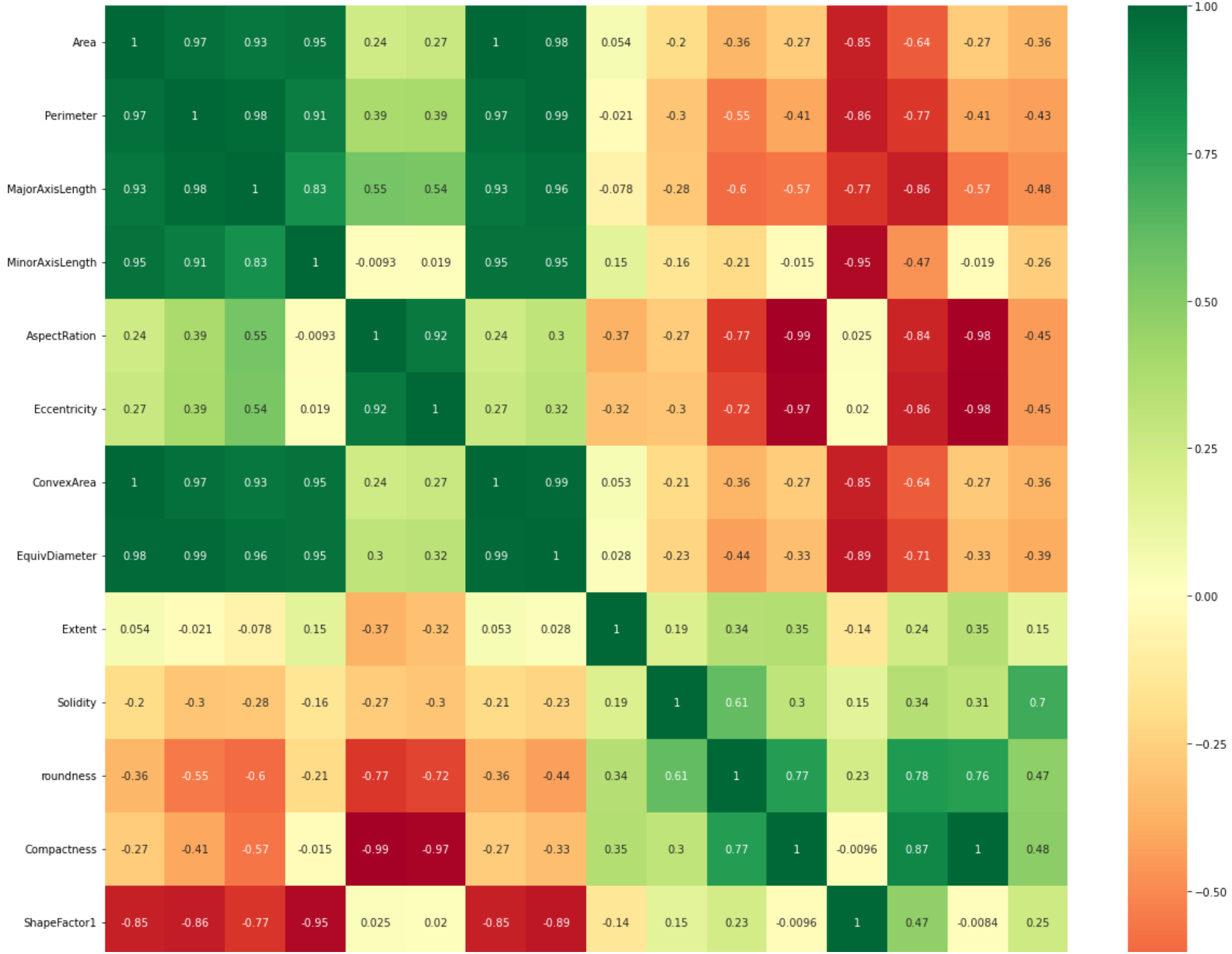
Out[8]:

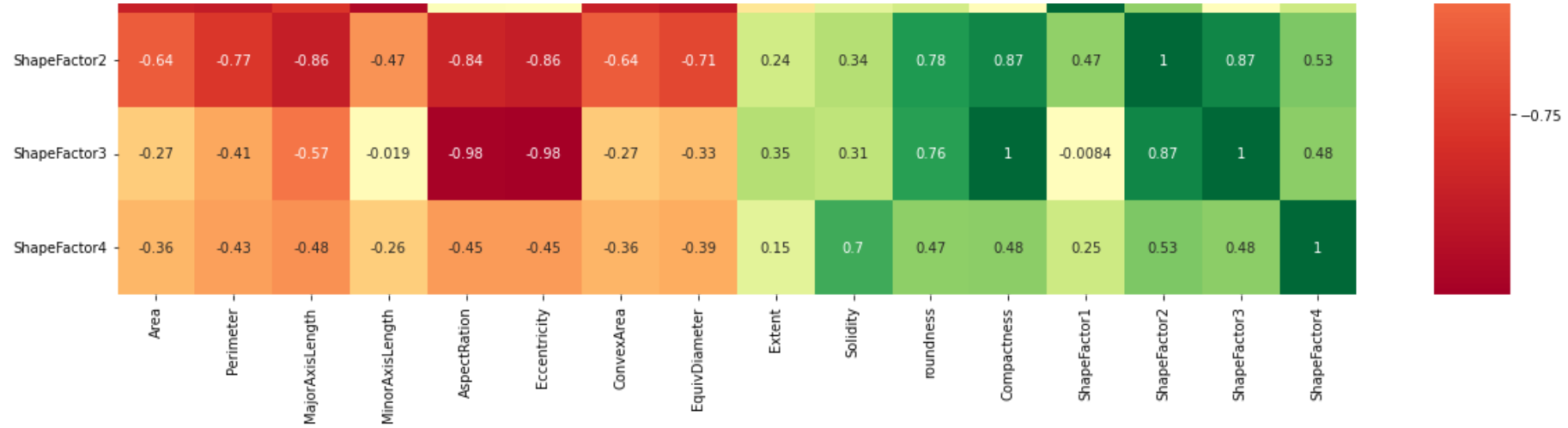
| | Bean ID | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent |
|---|---|---|---|---|---|---|---|---|---|---|
| **Bean ID** | 1.000000 | -0.369273 | -0.411175 | -0.325045 | -0.503044 | 0.139316 | 0.307748 | -0.369615 | -0.418614 | -0.099929 |
| **Area** | -0.369273 | 1.000000 | 0.966722 | 0.931836 | 0.951615 | 0.241692 | 0.267351 | 0.999939 | 0.984968 | 0.054345 |
| **Perimeter** | -0.411175 | 0.966722 | 1.000000 | 0.977337 | 0.913217 | 0.385228 | 0.390915 | 0.967689 | 0.991380 | -0.021160 |
| **MajorAxisLength** | -0.325045 | 0.931836 | 0.977337 | 1.000000 | 0.826119 | 0.550200 | 0.541746 | 0.932609 | 0.961732 | -0.077982 |
| **MinorAxisLength** | -0.503044 | 0.951615 | 0.913217 | 0.826119 | 1.000000 | -0.009318 | 0.019272 | 0.951353 | 0.948567 | 0.145987 |
| **AspectRation** | 0.139316 | 0.241692 | 0.385228 | 0.550200 | -0.009318 | 1.000000 | 0.924293 | 0.243258 | 0.303601 | -0.370102 |
| **Eccentricity** | 0.307748 | 0.267351 | 0.390915 | 0.541746 | 0.019272 | 0.924293 | 1.000000 | 0.269124 | 0.318512 | -0.319300 |
| **ConvexArea** | -0.369615 | 0.999939 | 0.967689 | 0.932609 | 0.951353 | 0.243258 | 0.269124 | 1.000000 | 0.985226 | 0.052564 |
| **EquivDiameter** | -0.418614 | 0.984968 | 0.991380 | 0.961732 | 0.948567 | 0.303601 | 0.318512 | 0.985226 | 1.000000 | 0.028383 |
| **Extent** | -0.099929 | 0.054345 | -0.021160 | -0.077982 | 0.145987 | -0.370102 | -0.319300 | 0.052564 | 0.028383 | 1.000000 |
| **Solidity** | 0.087452 | -0.196511 | -0.303887 | -0.284203 | -0.155750 | -0.267582 | -0.297427 | -0.206118 | -0.231566 | 0.191330 |
| **roundness** | 0.115936 | -0.357530 | -0.547647 | -0.596303 | -0.210290 | -0.766946 | -0.722177 | -0.362083 | -0.435945 | 0.344411 |
| **Compactness** | -0.200357 | -0.267840 | -0.406618 | -0.568118 | -0.014734 | -0.987689 | -0.970308 | -0.269698 | -0.327391 | 0.354102 |
| **ShapeFactor1** | 0.602935 | -0.847983 | -0.864671 | -0.773699 | -0.947207 | 0.024676 | 0.020108 | -0.847975 | -0.892783 | -0.141634 |
| **ShapeFactor2** | 0.074558 | -0.639305 | -0.767606 | -0.859237 | -0.471539 | -0.837749 | -0.859938 | -0.640876 | -0.713083 | 0.237880 |
| **ShapeFactor3** | -0.223753 | -0.272145 | -0.408435 | -0.568093 | -0.019175 | -0.978591 | -0.981057 | -0.274024 | -0.330389 | 0.347624 |
| **ShapeFactor4** | 0.058103 | -0.355721 | -0.429310 | -0.482477 | -0.263713 | -0.449225 | -0.449276 | -0.362049 | -0.392512 | 0.148502 |

In [9]:

```
corrmat  = df_tmp.corr()
#Correlation plot
top_corr_features = corrmat.index
plt.figure(figsize=(20,20))

#plot heat map
g=sns.heatmap(df_tmp[top_corr_features].corr(),annot=True,cmap="RdYlGn")
```


Area 1 0.97 0.93 0.95 0.24 0.27 1 0.98 0.054 -0.2 -0.36 -0.27 -0.85 -0.64 -0.27 -0.36
Perimeter 0.97 1 0.98 0.91 0.39 0.39 0.97 0.99 -0.021 -0.3 -0.55 -0.41 -0.86 -0.77 -0.41 -0.43
MajorAxisLength 0.93 0.98 1 0.83 0.55 0.54 0.93 0.96 -0.078 -0.28 -0.6 -0.57 -0.77 -0.86 -0.57 -0.48
MinorAxisLength 0.95 0.91 0.83 1 -0.0093 0.019 0.95 0.95 0.15 -0.16 -0.21 -0.015 -0.95 -0.47 -0.019 -0.26
AspectRation 0.24 0.39 0.55 -0.0093 1 0.92 0.24 0.3 -0.37 -0.27 -0.77 -0.99 0.025 -0.84 -0.98 -0.45
Eccentricity 0.27 0.39 0.54 0.019 0.92 1 0.27 0.32 -0.32 -0.3 -0.72 -0.97 0.02 -0.86 -0.98 -0.45
ConvexArea 1 0.97 0.93 0.95 0.24 0.27 1 0.99 0.053 -0.21 -0.36 -0.27 -0.85 -0.64 -0.27 -0.36
EquivDiameter 0.98 0.99 0.96 0.95 0.3 0.32 0.99 1 0.028 -0.23 -0.44 -0.33 -0.89 -0.71 -0.33 -0.39
Extent 0.054 -0.021 -0.078 0.15 -0.37 -0.32 0.053 0.028 1 0.19 0.34 0.35 -0.14 0.24 0.35 0.15
Solidity -0.2 -0.3 -0.28 -0.16 -0.27 -0.3 -0.21 -0.23 0.19 1 0.61 0.3 0.15 0.34 0.31 0.7
roundness -0.36 -0.55 -0.6 -0.21 -0.77 -0.72 -0.36 -0.44 0.34 0.61 1 0.77 0.23 0.78 0.76 0.47
Compactness -0.27 -0.41 -0.57 -0.015 -0.99 -0.97 -0.27 -0.33 0.35 0.3 0.77 1 -0.0096 0.87 1 0.48
ShapeFactor1 -0.85 -0.86 -0.77 -0.95 0.025 0.02 -0.85 -0.89 -0.14 0.15 0.23 -0.0096 1 0.47 -0.0084 0.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50




**PairPlot**

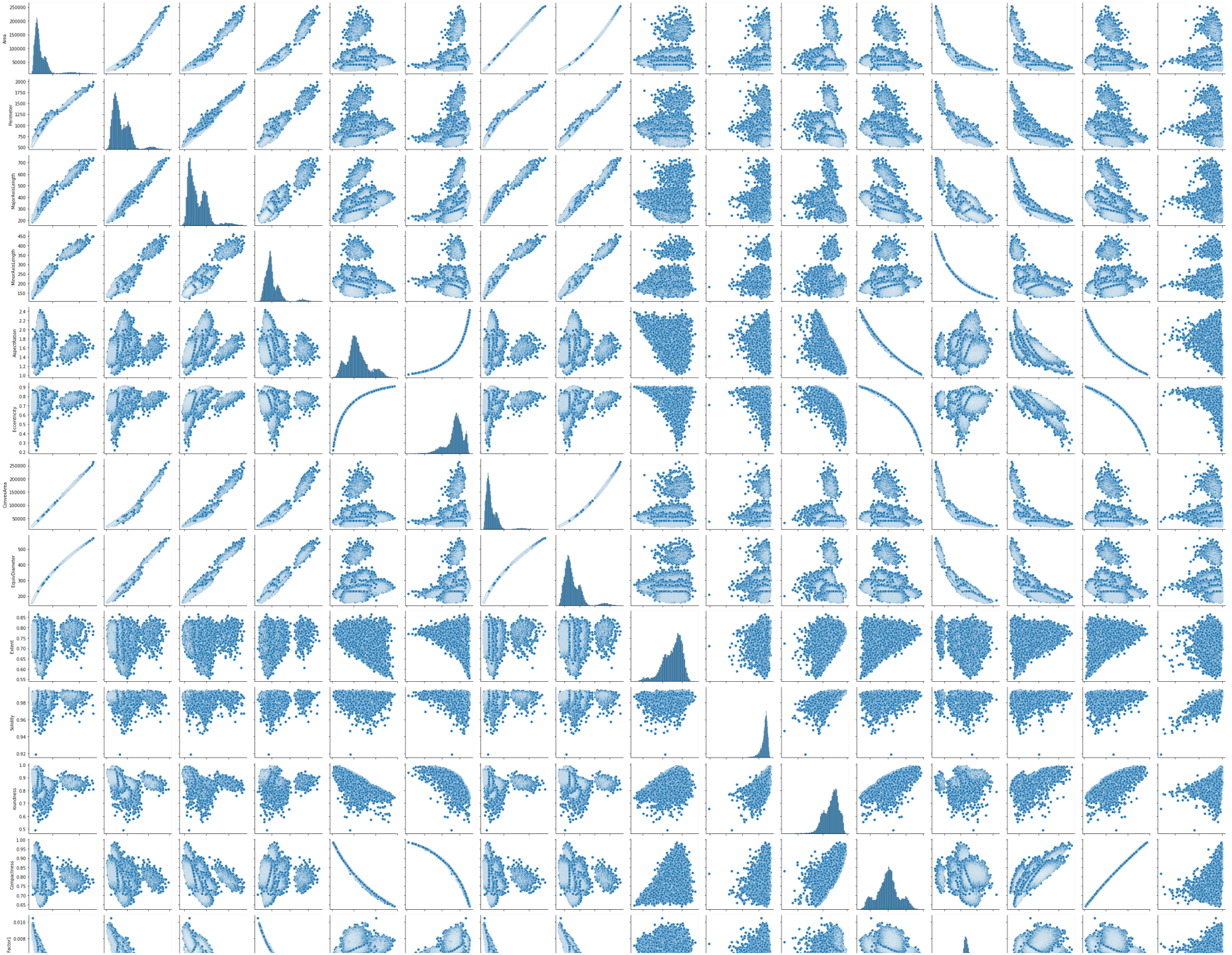
A pairplot plot a pairwise relationships in a dataset. The pairplot function creates a grid of Axes such that each variable in data will by shared in the y-axis across a single row and in the x-axis across a single column. That creates plots as shown below.

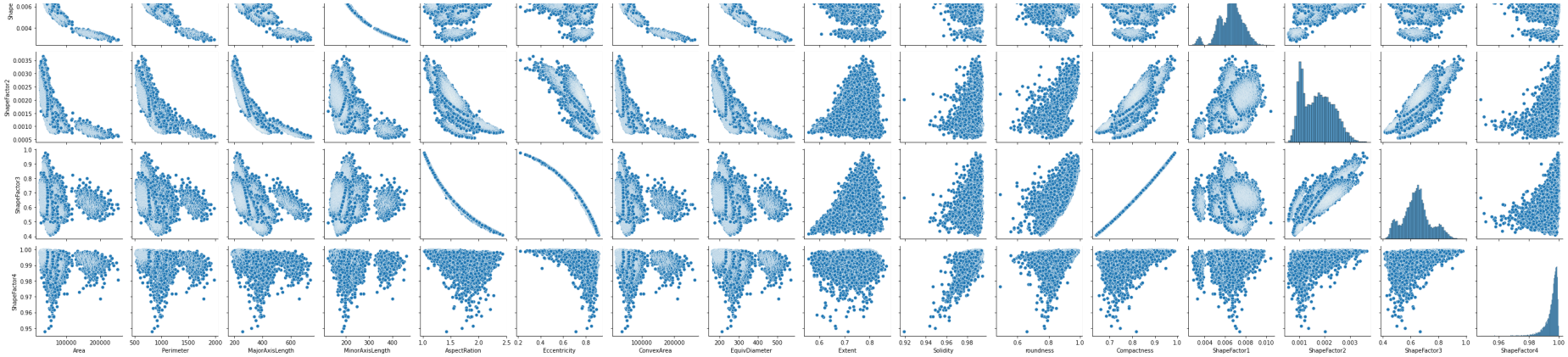
In [10]:
```
plt.figure(figsize = (40,40))

sns.pairplot(data=df_tmp)
```

Out[10]:
```
<seaborn.axisgrid.PairGrid at 0x1fe70769be0>

<Figure size 2880x2880 with 0 Axes>
```


Area
Perimeter
MajorAxisLength
MinorAxisLength
AspectRation
Eccentricity
ConvexArea
EquivDiameter
Extent
Solidity
roundness
Compactness
Factor1




It is clearly visible that there is multicollinearity between the features

Based on the heatmap and correlation data from above we can say that their is **MultiCollinearity** between the dependent features in the data

**MultiCollinearity** : Multicollinearity is a statistical concept where several independent variables in a model are correlated. Two variables are considered to be perfectly collinear if their correlation coefficient is +/- 1.0. Multicollinearity among independent variables will result in less reliable statistical inferences

Example : Area is highly correlated with ShapeFactor2,ConvexArea and EquiDiameter where ConvexArea and Equidiameter show positive correlation and ShapeFactor2 shows negative correlation|

## Feature Engineering

Feature engineering is the process of selecting, manipulating, and transforming raw data into features that can be used in supervised learning. In order to make machine learning work well on new tasks, it might be necessary to design and train better features. As you may know, a “feature” is any measurable input that can be used in a predictive model — it could be the color of an object or the sound of someone’s voice. Feature engineering, in simple terms, is the act of converting raw observations into desired features using statistical or machine learning approaches.

### Removing Unnecessary Data columns

Removing Rows that dont contribute to the learning of the model like Indexes in this dataset
The total sum of rows that have a missing value is 19.
When compared to the total rows(14,000) is very less. So we can drop the rows as the information loss will be minium

In [11]:
```
df_root.drop(columns= ["Bean ID"],inplace=True)
```

## NUll or Missing values

Handlind Missing or Null values in the dependent varialbles,

In [12]:
```
df_root.isna().sum()
```

Out[12]:
```
Area               0
Perimeter          0
MajorAxisLength    2
MinorAxisLength    3
AspectRation       1
Eccentricity       2
ConvexArea         0
EquivDiameter      0
Extent             0
Solidity           3
roundness          0
Compactness        4
ShapeFactor1       4
ShapeFactor2       2
ShapeFactor3       0
ShapeFactor4       0
Class              0
dtype: int64
```

**Number of rows will Null values are less and doesnt effect the distribution of the dataset, we can drop the rows**

After dropping the rows their is not no change in the statistical parameters of the data

In [13]:
```
df_root = df_root.dropna()
print(df_root.shape)
df_root.head()
```

```
(13591, 17)
```

Out[13]:

| | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent | Solidity | roundness | Com |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 28395.0 | 610.291 | 208.178117 | 173.888747 | 1.197191 | 0.549812 | 28715.0 | 190.141097 | 0.763923 | 0.988856 | 0.958027 | |
| **1** | 28734.0 | 638.018 | 200.524796 | 182.734419 | 1.097356 | 0.411785 | 29172.0 | 191.272750 | 0.783968 | 0.984986 | 0.887034 | |
| **2** | 29380.0 | 624.110 | 212.826130 | 175.931143 | 1.209713 | 0.562727 | 29690.0 | 193.410904 | 0.778113 | 0.989559 | 0.947849 | |
| **4** | 30140.0 | 620.134 | 201.847882 | 190.279279 | 1.060798 | 0.333680 | 30417.0 | 195.896503 | 0.773098 | 0.990893 | 0.984877 | |
| **5** | 30279.0 | 634.927 | 212.560556 | 181.510182 | 1.171067 | 0.520401 | 30600.0 | 196.347702 | 0.775688 | 0.989510 | 0.943852 | |

In [14]:
```
df_root.isna().sum()
```

Out[14]:
```
Area               0
Perimeter          0
MajorAxisLength    0
MinorAxisLength    0
AspectRation       0
Eccentricity       0
ConvexArea         0
EquivDiameter      0
Extent             0
Solidity           0
roundness          0
Compactness        0
ShapeFactor1       0
ShapeFactor2       0
ShapeFactor3       0
ShapeFactor4       0
Class              0
dtype: int64
```

In [15]:
```
df_root.describe()
```

Out[15]:

| | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent | |
|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 13591.000000 | 13591.000000 | 13591.000000 | 13591.000000 | 13591.000000 | 13591.000000 | 13591.000000 | 13591.000000 | 13591.000000 | 1359 |
| **mean** | 53077.569421 | 855.565981 | 320.281091 | 202.292563 | 1.583801 | 0.751189 | 53797.983298 | 253.134391 | 0.749698 | |
| **std** | 29335.656114 | 214.319365 | 85.679586 | 44.999344 | 0.246422 | 0.091731 | 29786.623118 | 59.192047 | 0.049109 | |
| **min** | 20420.000000 | 524.736000 | 183.601165 | 122.512653 | 1.024868 | 0.218951 | 20684.000000 | 161.243764 | 0.555315 | |
| **25%** | 36373.000000 | 703.897000 | 253.441135 | 175.792208 | 1.432988 | 0.716251 | 36750.000000 | 215.201166 | 0.718527 | |
| **50%** | 44674.000000 | 795.194000 | 297.079966 | 192.476741 | 1.551509 | 0.764576 | 45224.000000 | 238.496758 | 0.759785 | |
| **75%** | 61352.000000 | 977.360500 | 376.520088 | 217.124066 | 1.707447 | 0.810550 | 62326.500000 | 279.492026 | 0.786858 | |
| **max** | 254616.000000 | 1985.370000 | 738.860153 | 460.198497 | 2.430306 | 0.911423 | 263261.000000 | 569.374358 | 0.866195 | |

## Dealing with Target feature

Target Column is a categorical Column with 7 different classes.

As it is a classification problem we can map each class to a numerical value and replace in the dataset

In [16]:
```
dict_Class = {k:v for k, v in zip(list(set(df_root.iloc[:, -1].tolist())),
                                  list(range(len(list(set(df_root.iloc[:, -1].tolist()))))))}
dict_Class
```

Out[16]:
```
{'DERMASON': 0,
 'BOMBAY': 1,
 'CALI': 2,
 'BARBUNYA': 3,
 'HOROZ': 4,
 'SIRA': 5,
 'SEKER': 6}
```

In [17]:
```
df_root["Class"] = df_root["Class"].map(dict_Class)
df_root.head(8)
```

Out[17]:

| | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent | Solidity | roundness | Com |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 28395.0 | 610.291 | 208.178117 | 173.888747 | 1.197191 | 0.549812 | 28715.0 | 190.141097 | 0.763923 | 0.988856 | 0.958027 | |
| **1** | 28734.0 | 638.018 | 200.524796 | 182.734419 | 1.097356 | 0.411785 | 29172.0 | 191.272750 | 0.783968 | 0.984986 | 0.887034 | |
| **2** | 29380.0 | 624.110 | 212.826130 | 175.931143 | 1.209713 | 0.562727 | 29690.0 | 193.410904 | 0.778113 | 0.989559 | 0.947849 | |
| **4** | 30140.0 | 620.134 | 201.847882 | 190.279279 | 1.060798 | 0.333680 | 30417.0 | 195.896503 | 0.773098 | 0.990893 | 0.984877 | |
| **5** | 30279.0 | 634.927 | 212.560556 | 181.510182 | 1.171067 | 0.520401 | 30600.0 | 196.347702 | 0.775688 | 0.989510 | 0.943852 | |
| **6** | 30477.0 | 670.033 | 211.050155 | 184.039050 | 1.146768 | 0.489478 | 30970.0 | 196.988633 | 0.762402 | 0.984081 | 0.853080 | |
| **8** | 30685.0 | 635.681 | 213.534145 | 183.157146 | 1.165852 | 0.514081 | 31044.0 | 197.659696 | 0.771561 | 0.988436 | 0.954240 | |
| **9** | 30834.0 | 631.934 | 217.227813 | 180.897469 | 1.200834 | 0.553642 | 31120.0 | 198.139012 | 0.783683 | 0.990810 | 0.970278 | |

In [18]:
```
set(df_root.Class)
```

Out[18]:
```
{0, 1, 2, 3, 4, 5, 6}
```

## Check Distribution of Features

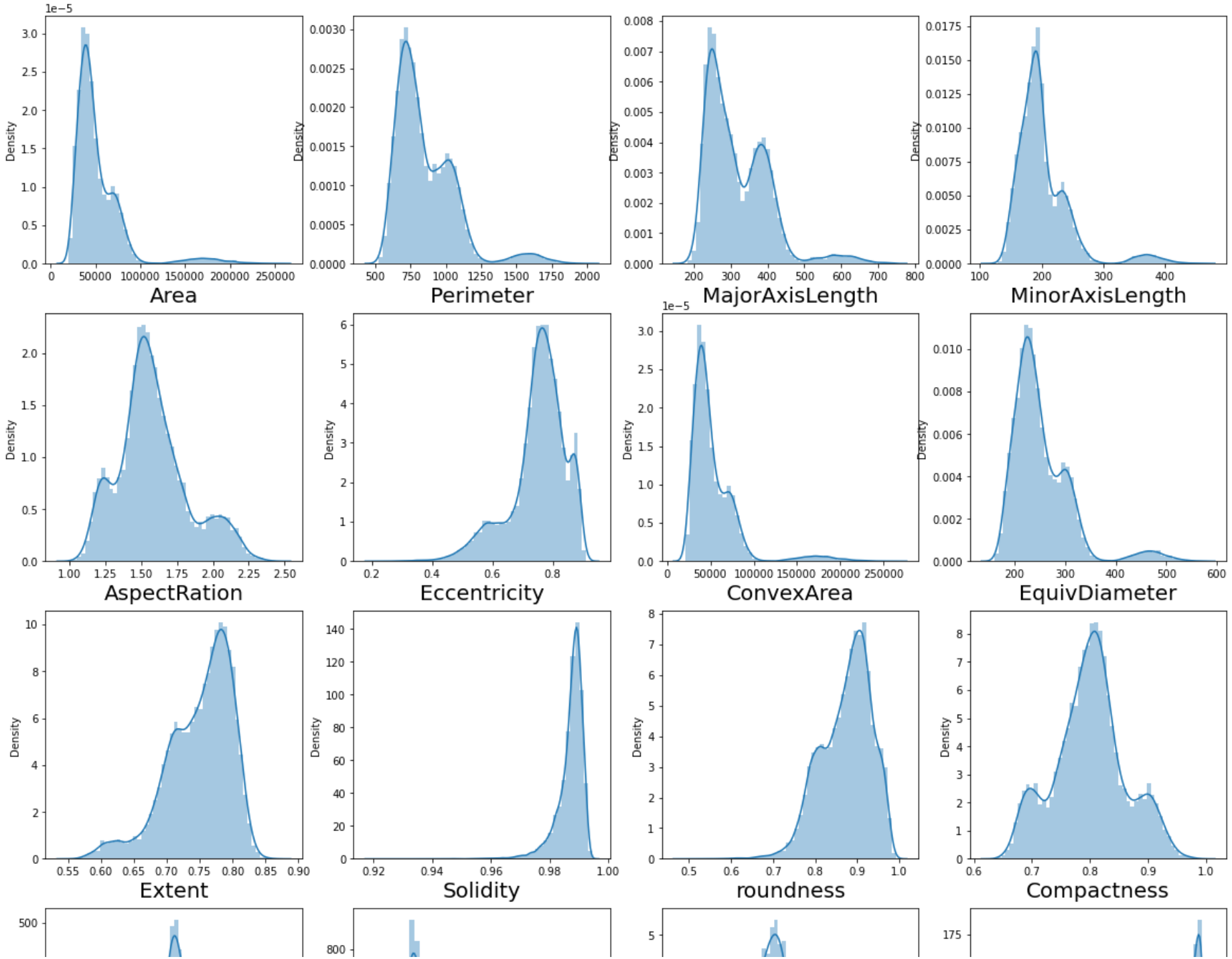
The distribution provides a parameterized mathematical function that can be used to calculate the probability for any individual observation from the sample space. This distribution describes the grouping or the density of the observations, called the probability density function. We can also calculate the likelihood of an observation having a value equal to or lesser than a given value. A summary of these relationships between observations is called a cumulative density function.

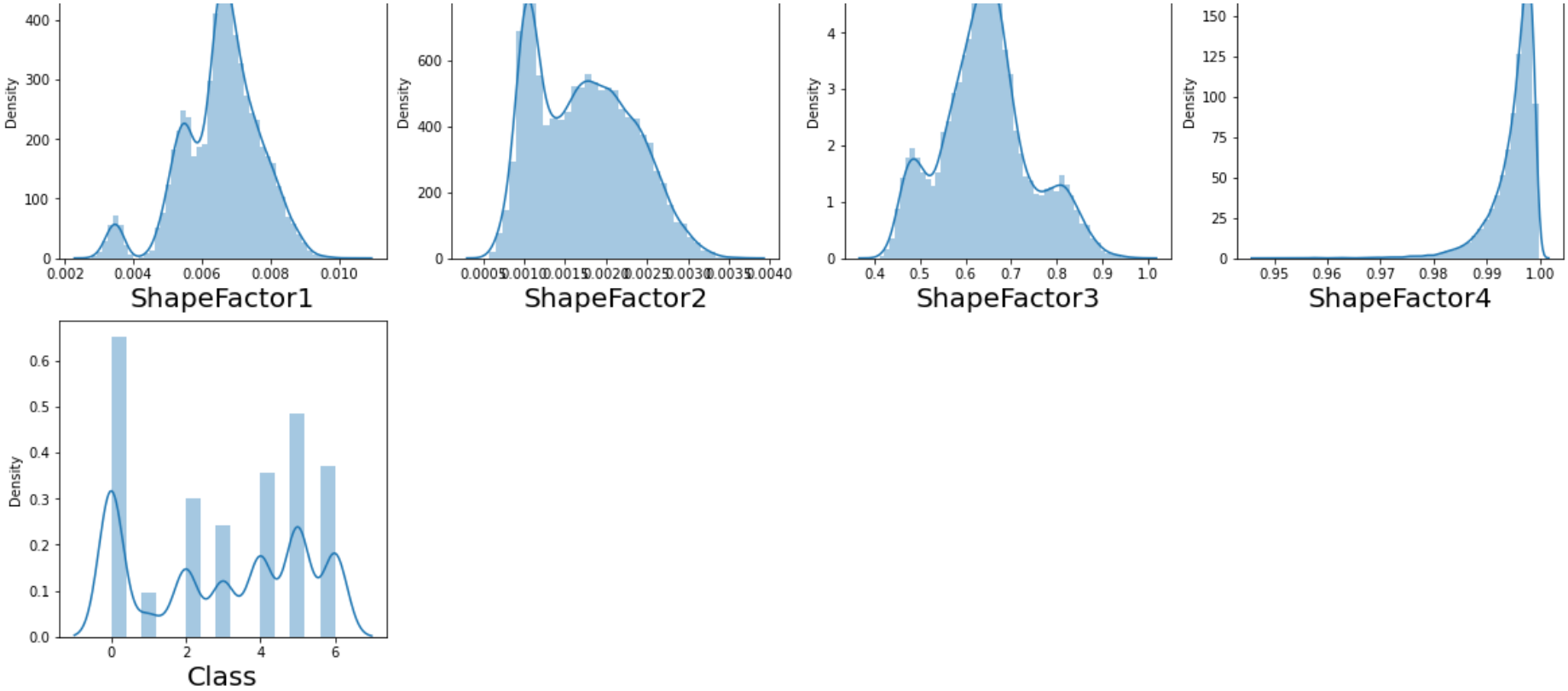
All the columns are almost follow a normal distributed so no need to Transformation of the data to make them Normally Distributed

In [19]:
```
# let's see how data is distributed for every column
df_tmp = df_root.copy()
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1
print(df_tmp.columns)
for column in df_tmp:
    if plotnumber<=18 :
        ax = plt.subplot(5,4,plotnumber)
        sns.distplot(df_tmp[column])
```

```
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.show()
```

```
Index(['Area', 'Perimeter', 'MajorAxisLength', 'MinorAxisLength',
       'AspectRation', 'Eccentricity', 'ConvexArea', 'EquivDiameter', 'Extent',
       'Solidity', 'roundness', 'Compactness', 'ShapeFactor1', 'ShapeFactor2',
       'ShapeFactor3', 'ShapeFactor4', 'Class'],
      dtype='object')
```


Area
Perimeter
MajorAxisLength
MinorAxisLength
AspectRation
Eccentricity
ConvexArea
EquivDiameter
Extent
Solidity
roundness
Compactness
Density
1e−5




**Most of the columns are normally distributed but some have skewness due to Outliers**

In [20]:
```
print("skewness of each column")
for col in df_root:
    print(col,"->",df_root[col].skew())
```

```
skewness of each column
Area -> 2.951255591634598
Perimeter -> 1.6249749765014962
MajorAxisLength -> 1.3576682685172654
MinorAxisLength -> 2.2357427158379135
AspectRation -> 0.5840957384596052
Eccentricity -> -1.065682818645653
ConvexArea -> 2.9401487885739988
EquivDiameter -> 1.947587451025246
Extent -> -0.8937382041461224
Solidity -> -2.550156750801783
roundness -> -0.6356652140347356
Compactness -> 0.03596081604829935
ShapeFactor1 -> -0.5328984958439338
ShapeFactor2 -> 0.29964487288456076
ShapeFactor3 -> 0.24173496282829024
ShapeFactor4 -> -2.758829845489117
Class -> -0.1576551412046186
```

## Outlier Handling

Outliers are those data points which differs significantly from other observations present in given dataset. It can occur because of variability in measurement and due to misinterpretation in filling data points.

In [21]:
```
df_outlier = df_root.drop(columns=["Class"])
df_outlier.head()
```

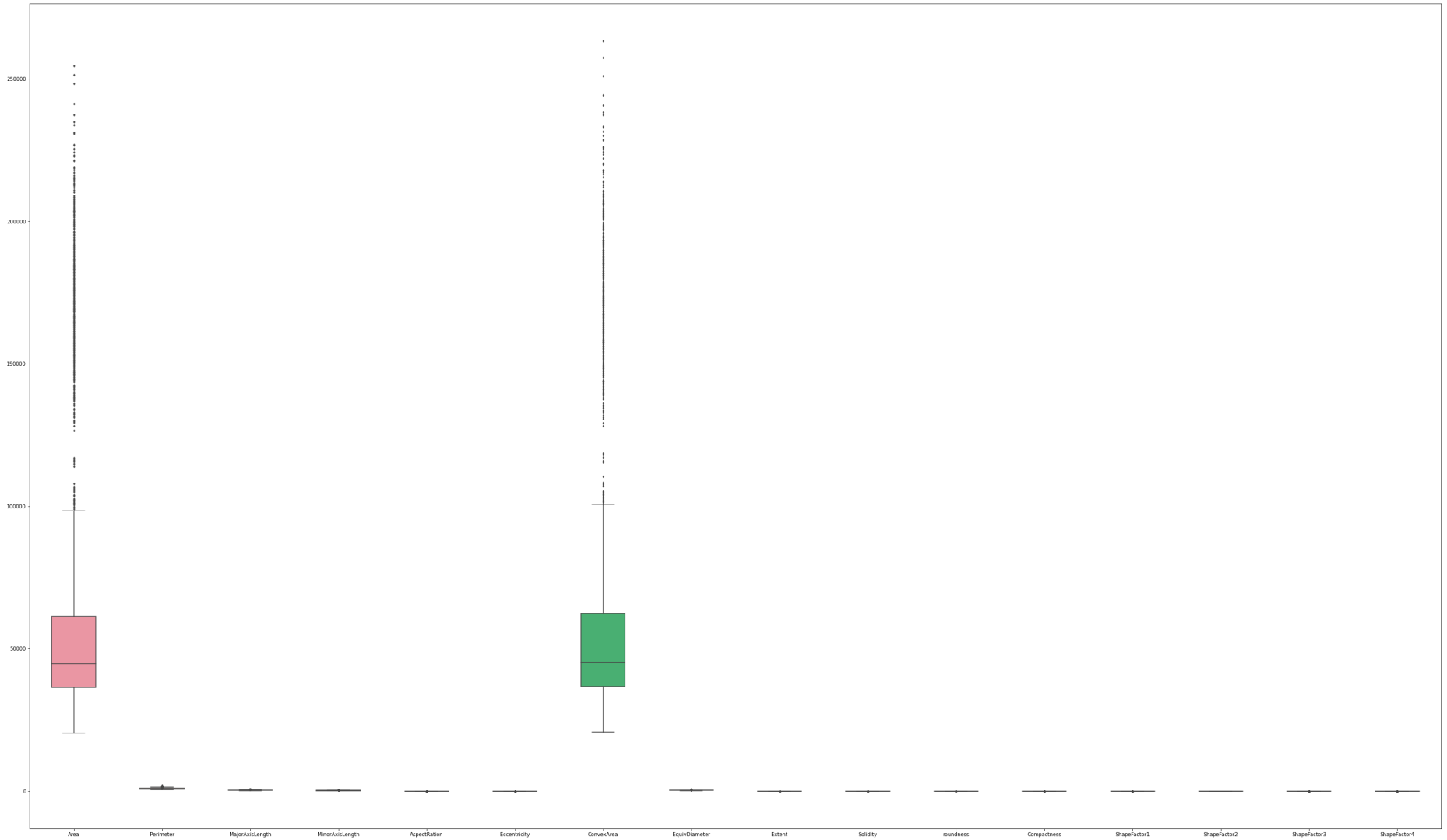
Out[21]:

| | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | EquivDiameter | Extent | Solidity | roundness | Com |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 28395.0 | 610.291 | 208.178117 | 173.888747 | 1.197191 | 0.549812 | 28715.0 | 190.141097 | 0.763923 | 0.988856 | 0.958027 | |
| **1** | 28734.0 | 638.018 | 200.524796 | 182.734419 | 1.097356 | 0.411785 | 29172.0 | 191.272750 | 0.783968 | 0.984986 | 0.887034 | |
| **2** | 29380.0 | 624.110 | 212.826130 | 175.931143 | 1.209713 | 0.562727 | 29690.0 | 193.410904 | 0.778113 | 0.989559 | 0.947849 | |
| **4** | 30140.0 | 620.134 | 201.847882 | 190.279279 | 1.060798 | 0.333680 | 30417.0 | 195.896503 | 0.773098 | 0.990893 | 0.984877 | |
| **5** | 30279.0 | 634.927 | 212.560556 | 181.510182 | 1.171067 | 0.520401 | 30600.0 | 196.347702 | 0.775688 | 0.989510 | 0.943852 | |

### Checking for outliers

In [22]:
```
fig, ax = plt.subplots(figsize=(50,30))
sns.boxplot(data=df_outlier, width= 0.5,ax=ax,  fliersize=3)
```

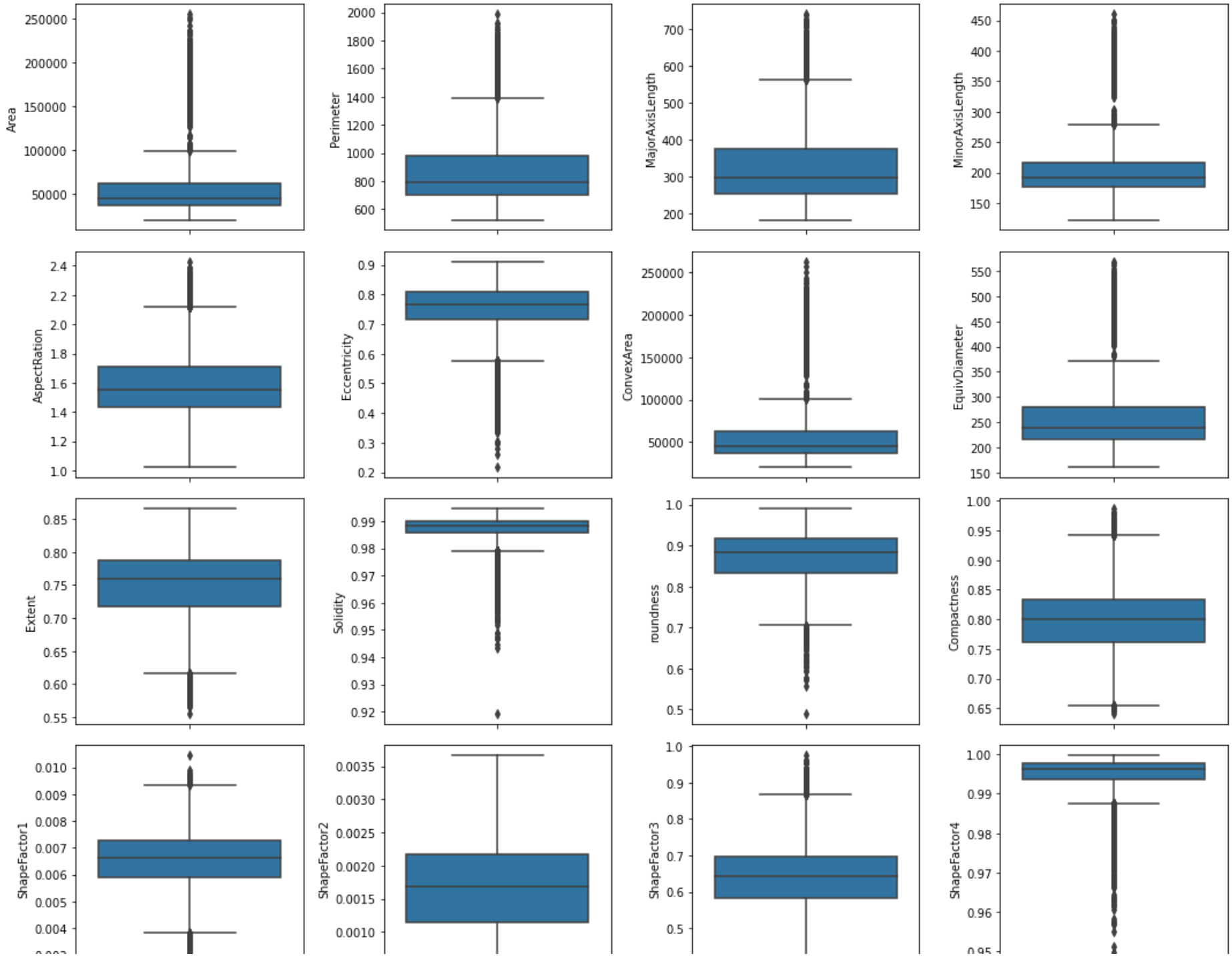
Out[22]:
```
<AxesSubplot:>
```




In [23]:
```
# Show BoxPlot for all features before outlier handling
Numeric_cols = df_outlier.columns

fig, ax = plt.subplots(4, 4, figsize=(15, 12))
```

```
for variable, subplot in zip(Numeric_cols, ax.flatten()):
    sns.boxplot(y= df_outlier[variable], ax=subplot)
plt.tight_layout()
```


Area
Perimeter
MajorAxisLength
MinorAxisLength
AspectRation
Eccentricity
ConvexArea
EquivDiameter
Extent
Solidity
roundness
Compactness
ShapeFactor1
ShapeFactor2
ShapeFactor3
ShapeFactor4


Many Outliers in the data espically in "Area" and "ConvexArea" columns

Option 1 : Deleting the outlier can cause a data loss of about 3000rows so to avoid that we can use some other method

Option 2 : Replace outlier with boundary of the distribution but as the count of the outlers is huge it will produce skewness in the data

Option 3 : So we can replace the outliers with the **Median** as median is not effected by the outliers

In [24]:
```
print("skewness of each column after Outlier Handling")
for column in df_outlier.columns:
    Q1= np.percentile(df_outlier[column], 25,interpolation = 'midpoint')
    Q3 = np.percentile(df_outlier[column], 75, interpolation = 'midpoint')
    IQR = Q3 - Q1

    median = df_outlier[column].quantile(0.50)
    df_outlier[column] = np.where(df_outlier[column] <=(Q1-1.5*IQR), median,df_outlier[column])
    df_outlier[column] = np.where(df_outlier[column] >=(Q3+1.5*IQR), median,df_outlier[column])
    print(column, "->", df_outlier[column].skew())
```

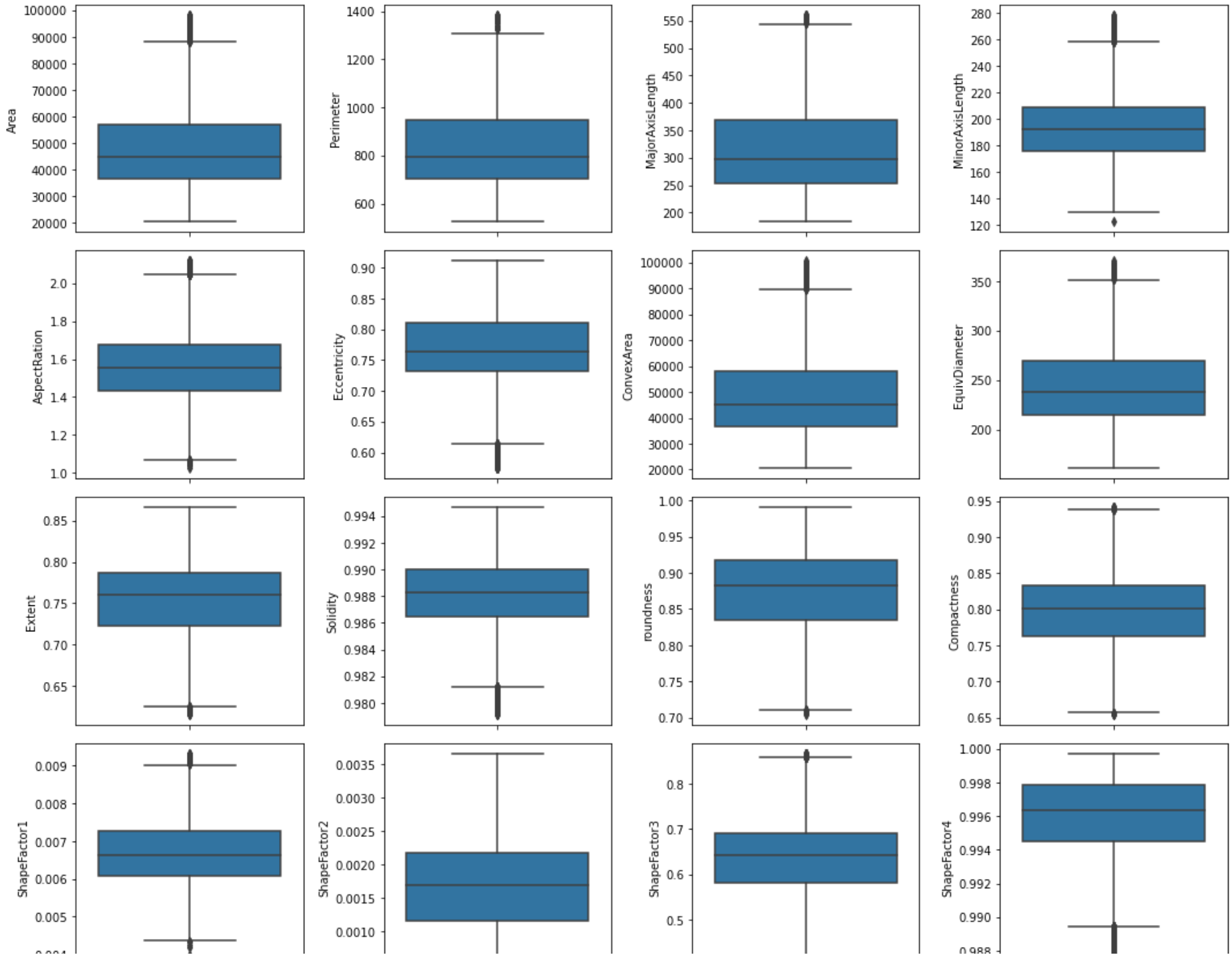
```
skewness of each column after Outlier Handling
Area -> 0.8307316808641193
Perimeter -> 0.597244434360571
MajorAxisLength -> 0.6381489693766639
MinorAxisLength -> 0.5637555378786383
AspectRation -> 0.3870834024559011
Eccentricity -> -0.476649885871401
ConvexArea -> 0.8362281840709223
EquivDiameter -> 0.5576262829534949
Extent -> -0.6504259333521553
Solidity -> -0.7714516398467093
roundness -> -0.42433582578985685
Compactness -> -0.016012129596301765
ShapeFactor1 -> 0.05792896804054203
ShapeFactor2 -> 0.29964487288456076
ShapeFactor3 -> 0.12236241153209534
ShapeFactor4 -> -1.0527209986865944
```

In [25]:
```
# Show BoxPlot for all features after outlier handling

Numeric_cols = df_outlier.columns

fig, ax = plt.subplots(4, 4, figsize=(15, 12))
```

```
for variable, subplot in zip(Numeric_cols, ax.flatten()):
    sns.boxplot(y= df_outlier[variable], ax=subplot)
plt.tight_layout()
```


Area
Perimeter
MajorAxisLength
MinorAxisLength
AspectRation
Eccentricity
ConvexArea
EquivDiameter
Extent
Solidity
roundness
Compactness
ShapeFactor1
ShapeFactor2
ShapeFactor3
ShapeFactor4


After Replacing the outliers with Median(robust to Outliers) the data looks good and outliers are lose of the boundaries which can be considered for building the data
Also the distribution of the data is also not skewed after removal of outliers

## Standardization and Normalization of Features

Feature scaling is one of the most important data preprocessing step in machine learning. Algorithms that compute the distance between the features are biased towards numerically larger values if the data is not scaled.

**Normalization or Min-Max Scaling** : is used to transform features to be on a similar scale. The new point is calculated as:

**Standardization or Z-Score Normalization** : is the transformation of features by subtracting from mean and dividing by standard deviation. This is often called as Z-score.

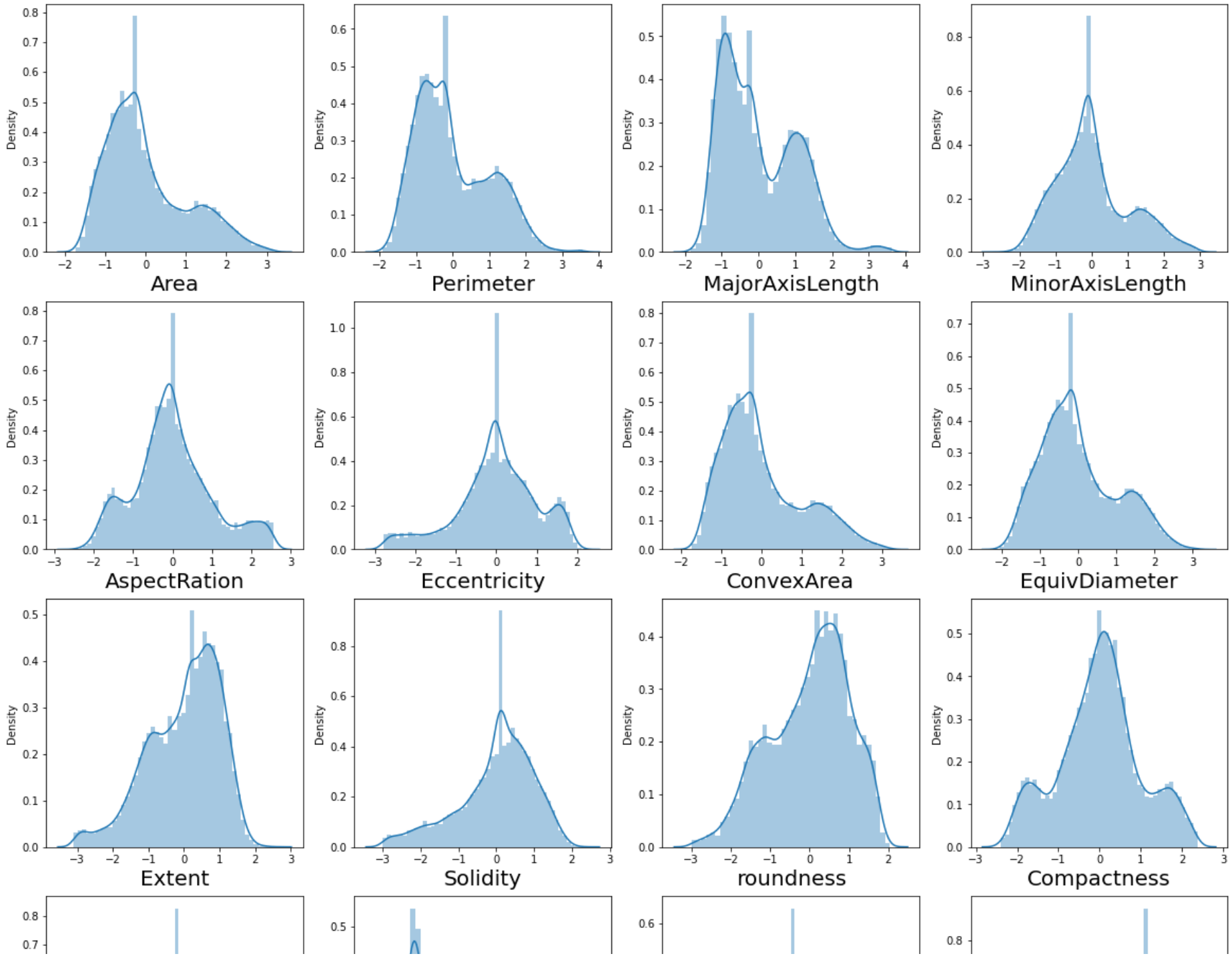
X_new = (X - mean)/Std
Standardization can be helpful in cases where the data follows a Gaussian distribution

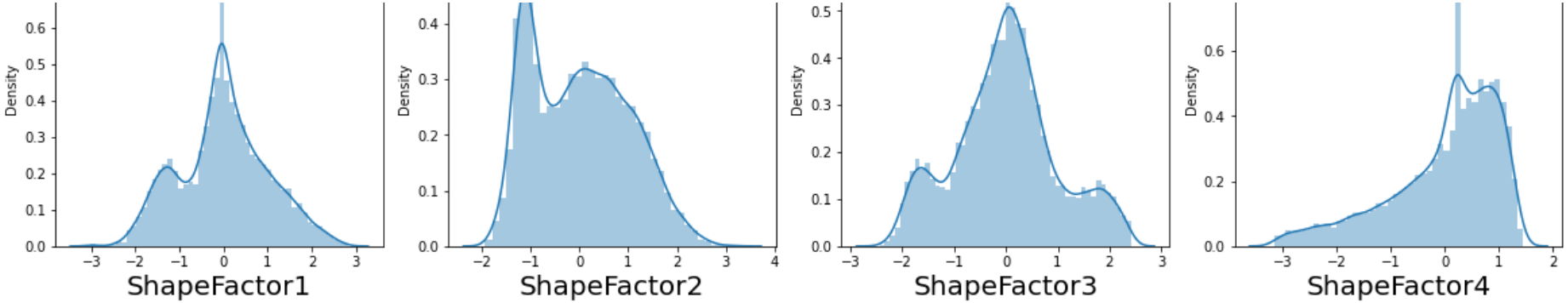
Data is normally Distributed so just applying Standard Scaling is enough

### Standard Scaling

In [26]:
```
trans = StandardScaler()
data = trans.fit_transform(df_outlier)
df_rootSS = pd.DataFrame(data,columns=df_outlier.columns)
```

In [27]:
```
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1
for column in df_rootSS.columns:
    if plotnumber<=18 :
        ax = plt.subplot(5,4,plotnumber)
        sns.distplot(df_rootSS[column])
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.show()
```

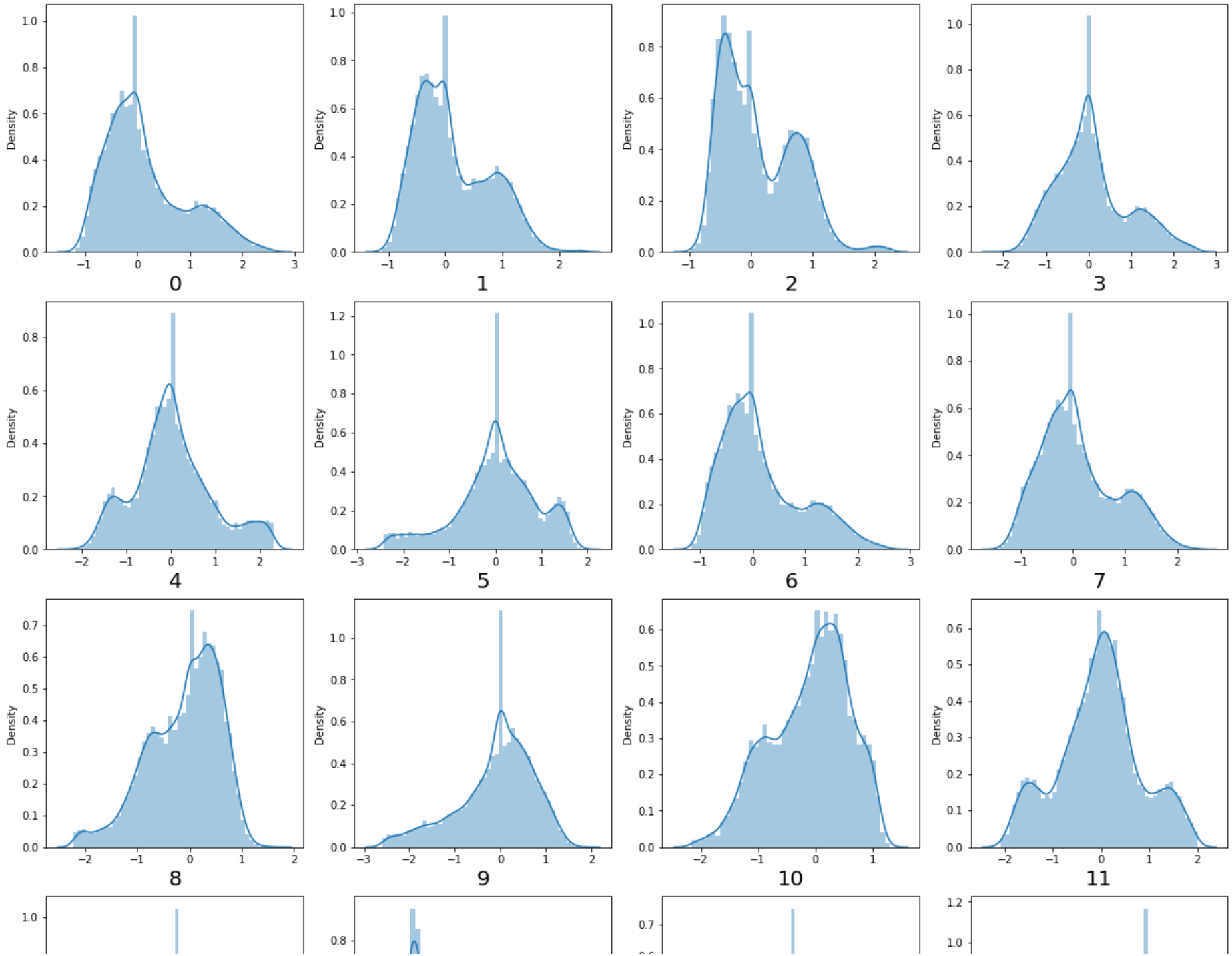

Density
Area
Perimeter
MajorAxisLength
MinorAxisLength
AspectRation
Eccentricity
ConvexArea
EquivDiameter
Extent
Solidity
roundness
Compactness

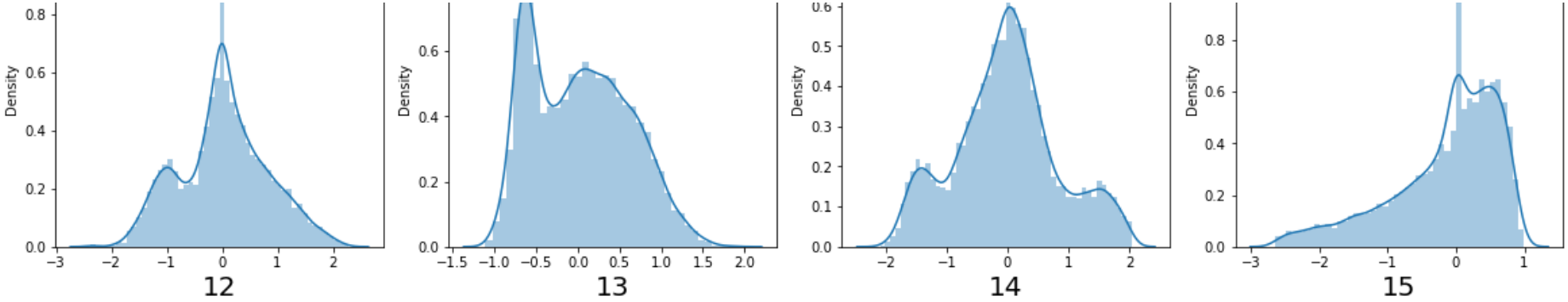



**Robust Scaling**

In [28]:
```
scalar = RobustScaler()
df_rootRB = pd.DataFrame(scalar.fit_transform(df_outlier))
```

In [29]:
```
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1
for column in df_rootRB.columns:
    if plotnumber<=18 :
        ax = plt.subplot(5,4,plotnumber)
        sns.distplot(df_rootRB[column])
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.show()
```


Density
0
1
2
3
4
5
6
7
8
9
10
11




In [30]:
```
dict_skew = {
    "Column" : df_rootSS.skew().index,
    "Standard Scaling" : df_rootSS.skew().values,
    "Robust Scaling" : df_rootRB.skew().values
}
skewness_df = pd.DataFrame(dict_skew)
skewness_df
```

Out[30]:

| | Column | Standard Scaling | Robust Scaling |
|---|---|---|---|
| **0** | Area | 0.830732 | 0.830732 |
| **1** | Perimeter | 0.597244 | 0.597244 |
| **2** | MajorAxisLength | 0.638149 | 0.638149 |
| **3** | MinorAxisLength | 0.563756 | 0.563756 |
| **4** | AspectRation | 0.387083 | 0.387083 |
| **5** | Eccentricity | -0.476650 | -0.476650 |
| **6** | ConvexArea | 0.836228 | 0.836228 |
| **7** | EquivDiameter | 0.557626 | 0.557626 |
| **8** | Extent | -0.650426 | -0.650426 |
| **9** | Solidity | -0.771452 | -0.771452 |
| **10** | roundness | -0.424336 | -0.424336 |
| **11** | Compactness | -0.016012 | -0.016012 |
| **12** | ShapeFactor1 | 0.057929 | 0.057929 |
| **13** | ShapeFactor2 | 0.299645 | 0.299645 |
| **14** | ShapeFactor3 | 0.122362 | 0.122362 |
| **15** | ShapeFactor4 | -1.052721 | -1.052721 |

**From the Table above we can see that both Transformation give same results and data is Normally Distributed so no other transformations are required**

Both Scaling techniques give same result, so we can use any of the scalling techniques here we went ahead with Standard Scaler

In [31]:
```
df_prime = df_rootSS.copy()
df_prime["Class"] = list(df_root["Class"])
```

## Feature Selection

Feature selection is a way of selecting the subset of the most relevant features from the original features set by removing the redundant, irrelevant, or noisy features.

## MultiCollinearlity(vif)

**Variance Inflation Factor** : A variance inflation factor(VIF) detects multicollinearity in regression analysis. Multicollinearity is when there's correlation between predictors (i.e. independent variables) in a model; it's presence can adversely affect your regression results. The VIF estimates how much the variance of a regression coefficient is inflated due to multicollinearity in the model.

$$\text{VIF} = \frac{1}{1 - R_i^2}$$

We need to remove the columns with high VIF score so that score of all features gets below 5
So we need to remove Feature with high VIFs and again calculate for all the rest of the columns and continue to do so until all the columns score is around 5

In [32]:
```
def vif_score(x):
    scalar = StandardScaler()
    arr = scalar.fit_transform(x)
    return pd.DataFrame([[x.columns[i], variance_inflation_factor(arr,i)] for i in range(arr.shape[1])], columns = ["Features","V

vif_score(df_prime)
```

Out[32]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 987.574141 |
| **1** | Perimeter | 37.668257 |
| **2** | MajorAxisLength | 10.226559 |
| **3** | MinorAxisLength | 28.718741 |
| **4** | AspectRation | 5.676173 |
| **5** | Eccentricity | 4.469610 |
| **6** | ConvexArea | 1019.261547 |
| **7** | EquivDiameter | 106.526744 |
| **8** | Extent | 1.120211 |
| **9** | Solidity | 1.895040 |
| **10** | roundness | 6.262355 |
| **11** | Compactness | 29.402958 |
| **12** | ShapeFactor1 | 23.336771 |
| **13** | ShapeFactor2 | 10.740645 |
| **14** | ShapeFactor3 | 23.213154 |
| **15** | ShapeFactor4 | 1.660883 |
| **16** | Class | 1.514598 |

In [33]:
```
df_tmp = df_prime.copy()
```

In [34]:
```
df_tmp.drop(columns=['Class'],inplace=True)
vif_score(df_tmp)
```

Out[34]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 987.129191 |
| **1** | Perimeter | 37.507568 |
| **2** | MajorAxisLength | 10.226324 |
| **3** | MinorAxisLength | 28.634145 |
| **4** | AspectRation | 5.676114 |
| **5** | Eccentricity | 4.415332 |
| **6** | ConvexArea | 1014.440900 |
| **7** | EquivDiameter | 105.643845 |
| **8** | Extent | 1.117034 |
| **9** | Solidity | 1.863884 |
| **10** | roundness | 6.125728 |
| **11** | Compactness | 29.099187 |
| **12** | ShapeFactor1 | 20.434658 |
| **13** | ShapeFactor2 | 10.278319 |
| **14** | ShapeFactor3 | 23.203049 |
| **15** | ShapeFactor4 | 1.660862 |

In [35]:

```
df_tmp.drop(columns=['ConvexArea'],inplace=True)
vif_score(df_tmp)
```

Out[35]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 44.239569 |
| **1** | Perimeter | 37.261501 |
| **2** | MajorAxisLength | 10.226292 |
| **3** | MinorAxisLength | 28.441701 |
| **4** | AspectRation | 5.673256 |
| **5** | Eccentricity | 4.413360 |
| **6** | EquivDiameter | 105.241687 |
| **7** | Extent | 1.116506 |
| **8** | Solidity | 1.830235 |
| **9** | roundness | 6.110601 |
| **10** | Compactness | 29.095556 |
| **11** | ShapeFactor1 | 20.242795 |
| **12** | ShapeFactor2 | 10.255677 |
| **13** | ShapeFactor3 | 23.202849 |
| **14** | ShapeFactor4 | 1.660845 |

In [36]:

```
df_tmp.drop(columns=['EquivDiameter'],inplace=True)
vif_score(df_tmp)
```

Out[36]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 26.377697 |
| **1** | Perimeter | 30.865171 |
| **2** | MajorAxisLength | 10.133648 |
| **3** | MinorAxisLength | 25.818747 |
| **4** | AspectRation | 5.632993 |
| **5** | Eccentricity | 4.412026 |
| **6** | Extent | 1.116506 |
| **7** | Solidity | 1.821931 |
| **8** | roundness | 6.071027 |
| **9** | Compactness | 28.416524 |
| **10** | ShapeFactor1 | 18.196214 |
| **11** | ShapeFactor2 | 10.164802 |
| **12** | ShapeFactor3 | 23.179403 |
| **13** | ShapeFactor4 | 1.660495 |

In [37]:

```
df_tmp.drop(columns=['Perimeter'],inplace=True)
vif_score(df_tmp)
```

Out[37]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 20.837077 |
| **1** | MajorAxisLength | 8.639152 |
| **2** | MinorAxisLength | 25.759283 |
| **3** | AspectRation | 5.632554 |
| **4** | Eccentricity | 4.410109 |
| **5** | Extent | 1.114657 |
| **6** | Solidity | 1.821883 |
| **7** | roundness | 4.941783 |
| **8** | Compactness | 28.276408 |
| **9** | ShapeFactor1 | 16.614149 |
| **10** | ShapeFactor2 | 10.123211 |
| **11** | ShapeFactor3 | 23.176231 |
| **12** | ShapeFactor4 | 1.659389 |

In [38]:

```
df_tmp.drop(columns=['Compactness'],inplace=True)
vif_score(df_tmp)
```

Out[38]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 20.170413 |
| **1** | MajorAxisLength | 8.635127 |
| **2** | MinorAxisLength | 25.306934 |
| **3** | AspectRation | 5.353142 |
| **4** | Eccentricity | 4.392590 |
| **5** | Extent | 1.113670 |
| **6** | Solidity | 1.812668 |
| **7** | roundness | 4.883257 |
| **8** | ShapeFactor1 | 16.410663 |
| **9** | ShapeFactor2 | 9.602287 |
| **10** | ShapeFactor3 | 11.323253 |
| **11** | ShapeFactor4 | 1.658743 |

In [39]:

```
df_tmp.drop(columns=['MinorAxisLength'],inplace=True)
vif_score(df_tmp)
```

Out[39]:

| | Features | VIF_Score |
|---|---|---|
| **0** | Area | 12.622901 |
| **1** | MajorAxisLength | 8.574261 |
| **2** | AspectRation | 5.305245 |
| **3** | Eccentricity | 4.392573 |
| **4** | Extent | 1.113173 |
| **5** | Solidity | 1.802818 |
| **6** | roundness | 4.869437 |
| **7** | ShapeFactor1 | 8.727018 |
| **8** | ShapeFactor2 | 9.581197 |
| **9** | ShapeFactor3 | 11.006606 |
| **10** | ShapeFactor4 | 1.656902 |

In [40]:

```
df_tmp.drop(columns=['Area'],inplace=True)
vif_score(df_tmp)
```

Out[40]:

| | Features | VIF_Score |
|---|---|---|
| **0** | MajorAxisLength | 7.322690 |
| **1** | AspectRation | 5.170812 |
| **2** | Eccentricity | 4.387638 |
| **3** | Extent | 1.112438 |
| **4** | Solidity | 1.791607 |
| **5** | roundness | 4.790035 |
| **6** | ShapeFactor1 | 3.378236 |
| **7** | ShapeFactor2 | 9.444791 |
| **8** | ShapeFactor3 | 10.625017 |
| **9** | ShapeFactor4 | 1.655286 |

In [41]:
```
df_tmp.drop(columns=['ShapeFactor3'],inplace=True)
vif_score(df_tmp)
```

Out[41]:

| | Features | VIF_Score |
|---|---|---|
| **0** | MajorAxisLength | 7.183420 |
| **1** | AspectRation | 4.700802 |
| **2** | Eccentricity | 3.190545 |
| **3** | Extent | 1.111899 |
| **4** | Solidity | 1.781582 |
| **5** | roundness | 4.555491 |
| **6** | ShapeFactor1 | 2.768339 |
| **7** | ShapeFactor2 | 8.582353 |
| **8** | ShapeFactor4 | 1.652437 |

In [42]:
```
df_tmp.drop(columns=['ShapeFactor2'],inplace=True)
```

```
vif_score(df_tmp)
```

Out[42]:

| | Features | VIF_Score |
|---|---|---|
| **0** | MajorAxisLength | 5.478921 |
| **1** | AspectRation | 3.795605 |
| **2** | Eccentricity | 3.179796 |
| **3** | Extent | 1.108359 |
| **4** | Solidity | 1.744490 |
| **5** | roundness | 4.142303 |
| **6** | ShapeFactor1 | 2.766302 |
| **7** | ShapeFactor4 | 1.574653 |

In [43]:
```
vif_var = vif_score(df_tmp)
colms = (vif_var["Features"].values)
df_prime = df_prime[np.append(colms,"Class")]
```

In [44]:
```
df_prime.head()
```

Out[44]:

| | MajorAxisLength | AspectRation | Eccentricity | Extent | Solidity | roundness | ShapeFactor1 | ShapeFactor4 | Class |
|---|---|---|---|---|---|---|---|---|---|
| **0** | -1.494114 | -1.673551 | -0.025058 | 0.249564 | 0.300339 | 1.467362 | 0.697402 | 1.072735 | 6 |
| **1** | -1.604958 | -2.132047 | -0.025058 | 0.703317 | -1.018448 | 0.217883 | 0.320377 | 0.962874 | 6 |
| **2** | -1.426796 | -1.616047 | -0.025058 | 0.570786 | 0.539801 | 1.288237 | 0.603806 | 1.200817 | 6 |
| **3** | -1.585795 | -2.299943 | -0.025058 | 0.457261 | 0.994507 | 1.939919 | 0.019427 | 1.238212 | 6 |
| **4** | -1.430643 | -1.793529 | -0.025058 | 0.515899 | 0.523115 | 1.217878 | 0.364620 | 1.264304 | 6 |

**MultiCollinearily removed**

## HeatMap Correlation

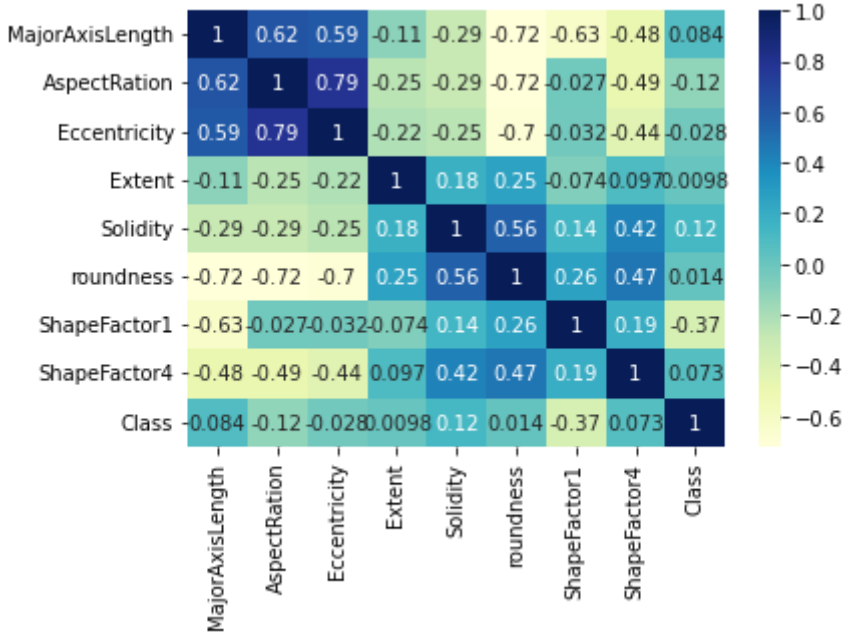
In [45]:
```
print(df_prime.corr())
```

```
sns.heatmap(df_prime.corr(), cmap="YlGnBu", annot=True)
```

```
                 MajorAxisLength  AspectRation  Eccentricity    Extent  \
MajorAxisLength         1.000000      0.623954      0.594229 -0.113960
AspectRation            0.623954      1.000000      0.790845 -0.253065
Eccentricity            0.594229      0.790845      1.000000 -0.218750
Extent                 -0.113960     -0.253065     -0.218750  1.000000
Solidity               -0.291867     -0.292597     -0.254339  0.178352
roundness              -0.718971     -0.716648     -0.695146  0.254880
ShapeFactor1           -0.628504     -0.026844     -0.031794 -0.074377
ShapeFactor4           -0.484125     -0.486231     -0.435788  0.096686
Class                   0.083533     -0.124493     -0.027616  0.009776

                 Solidity  roundness  ShapeFactor1  ShapeFactor4     Class
MajorAxisLength -0.291867  -0.718971     -0.628504     -0.484125  0.083533
AspectRation    -0.292597  -0.716648     -0.026844     -0.486231 -0.124493
Eccentricity    -0.254339  -0.695146     -0.031794     -0.435788 -0.027616
Extent           0.178352   0.254880     -0.074377      0.096686  0.009776
Solidity         1.000000   0.557045      0.135051      0.415943  0.120401
roundness        0.557045   1.000000      0.263692      0.471580  0.013536
ShapeFactor1     0.135051   0.263692      1.000000      0.193376 -0.370859
ShapeFactor4     0.415943   0.471580      0.193376      1.000000  0.072610
Class            0.120401   0.013536     -0.370859      0.072610  1.000000
```

Out[45]: <AxesSubplot:>

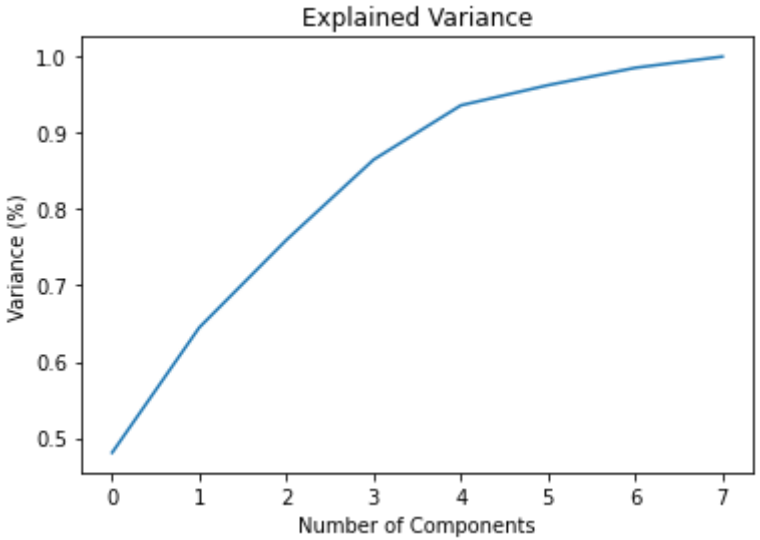



**There is no high correlation between features and target variables**

## Principle Component Analysis

Principal Component Analysis is an unsupervised learning algorithm that is used for the **dimensionality decomposition** in machine learning. It is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. These new transformed features are called the Principal Components

In [46]:
```
pca = PCA()
principalComponents = pca.fit_transform(df_prime.drop(columns=["Class"]))
plt.figure()
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Variance (%)') #for each component
plt.title('Explained Variance')
plt.show()
```




From the diagram above, it can be seen that 4 principal components explain almost 90% of the variance in data and 6 principal components explain around 95% of the variance in data.

So, instead of giving all the columns as input, we'd only feed these 5 principal components of the data to the machine learning algorithm and we'd obtain a similar result.

In [47]:
```
pca = PCA(n_components=5)
new_data = pca.fit_transform(df_prime.drop(columns=["Class"]))
# This will be the new data fed to the algorithm.
principal_Df = pd.DataFrame(data = new_data
             , columns = ['principal component 1', 'principal component 2','principal component 3','principal component 4','princ
principal_Df['Class'] = list(df_prime["Class"])
principal_Df
```

Out[47]:

| | principal component 1 | principal component 2 | principal component 3 | principal component 4 | principal component 5 | Class |
|---|---|---|---|---|---|---|
| **0** | -2.655565 | -0.497196 | 0.148967 | 0.136680 | -0.270488 | 6 |
| **1** | -1.916533 | -0.017437 | 1.229663 | 0.852133 | -1.178407 | 6 |
| **2** | -2.663991 | -0.277538 | -0.180201 | 0.198720 | -0.432386 | 6 |
| **3** | -3.343154 | 0.255319 | -0.204249 | -0.251362 | -0.000988 | 6 |
| **4** | -2.675149 | -0.087268 | -0.069893 | 0.050199 | -0.487487 | 6 |
| **...** | ... | ... | ... | ... | ... | ... |
| **13586** | -0.958646 | -0.624323 | -0.386779 | -1.174194 | -0.005368 | 0 |
| **13587** | -1.653223 | 0.543385 | -0.924785 | 0.178687 | -0.296305 | 0 |
| **13588** | -1.129661 | -0.140446 | 0.005781 | -0.691203 | 0.427898 | 0 |
| **13589** | -0.482116 | -0.461576 | 0.769115 | -0.613146 | 0.705906 | 0 |
| **13590** | -0.621267 | -0.120633 | -1.079679 | 0.248793 | -0.677759 | 0 |

13591 rows × 6 columns

Number of columns are decreased from 18 to 8 usinf ViF scores and from 8 to 5 using Principle Component Analysis(Curse of Dimensionality reduced)

### Split Independent and Dependent Features

In [49]:
```
X = principal_Df.drop(columns = ["Class"])
y = principal_Df.Class
```

## Case 1 : 80:20

### Train Test Split

The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model.

Split the data into Training and test data where 80% of the data is used for learning the model and test 20% is used for testing/validation

In [50]:
```
x_train_c1,x_test_c1,y_train_c1,y_test_c1 = train_test_split(X,y, test_size= 0.20, random_state = 355)
```

In [51]:
```
x_train_c1.shape
```

Out[51]:
```
(10872, 5)
```

In [52]:
```
x_test_c1.shape
```

Out[52]:
```
(2719, 5)
```

In [53]:
```
print(y_train_c1.shape)
{ k:list(y_train_c1).count(k) for k in set(y_train_c1)}
```

Out[53]:
```
(10872,)
{0: 2846, 1: 435, 2: 1291, 3: 1068, 4: 1555, 5: 2096, 6: 1581}
```

In [54]:
```
print(y_test_c1.shape)
{ k:list(y_test_c1).count(k) for k in set(y_test_c1)}
```

Out[54]:
```
(2719,)
{0: 700, 1: 87, 2: 339, 3: 254, 4: 373, 5: 540, 6: 426}
```

Checking if after the split one class is dominating the other class or not

Data us well split and balanced

# Model Building

## Logestic Regression

Need to select parameters as multinormial or OVR(one verses Rest)

In [55]:
```
lr_model_c1 = LogisticRegression(multi_class='multinomial',)
lr_model_c1.fit(x_train_c1,y_train_c1)
```

Out[55]:
```
LogisticRegression(multi_class='multinomial')
```

### Random Forest

Compare LR model with a none Linear Model

In [56]:
```
rf_classifier_c1 = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 42)
rf_classifier_c1.fit(x_train_c1,y_train_c1)
```

Out[56]:
```
RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=42)
```

### K fold Cross Validation

It is a data partitioning strategy so that you can effectively use your dataset to build a more generalized model. The main intention of doing any kind of machine learning is to develop a more generalized model which can perform well on unseen data. One can build a perfect model on the training data with 100% accuracy or 0 error, but it may fail to generalize for unseen data. So, it is not a good model. It overfits the training data. Machine Learning is all about generalization meaning that model's performance can only be measured with data points that have never been used during the training process. That is why we often split our data into a training set and a test set.

In [88]:
```
kfold_validation=KFold(40)
results_kfold_lr_c1 = cross_val_score(lr_model_c1,X,y,cv=kfold_validation)
results_kfold_rf_c1 = cross_val_score(rf_classifier_c1,X,y,cv=kfold_validation)
print("K_Fold validation Score Logestic Regression(Linear Model) CASE1(80:20): ",np.mean(results_kfold_lr_c1))
print("K_Fold validation Score Random Forest (Non - Linear Model) CASE1(80:20): ",np.mean(results_kfold_rf_c1))
```

```
K_Fold validation Score Logestic Regression(Linear Model) CASE1(80:20):  0.8070338799236509
K_Fold validation Score Random Forest (Non - Linear Model) CASE1(80:20):  0.7896800711435017
```

After the kfold Validation we can see that the models is well build and generalized.
After Comparision with a nonLinear model the Linear Model performs well

## Case 2 : 90:10

### Train Test Split

In [57]:
```
x_train_c2,x_test_c2,y_train_c2,y_test_c2 = train_test_split(X,y, test_size= 0.10, random_state = 355)
```

In [58]:
```
x_train_c2.shape
```

Out[58]:
```
(12231, 5)
```

In [59]:
```
x_test_c2.shape
```

Out[59]:
```
(1360, 5)
```

In [60]:
```
print(y_train_c2.shape)
{ k:list(y_train_c2).count(k) for k in set(y_train_c2)}
```

Out[60]:
```
(12231,)
{0: 3200, 1: 475, 2: 1462, 3: 1182, 4: 1750, 5: 2374, 6: 1788}
```

In [61]:
```
print(y_test_c2.shape)
{ k:list(y_test_c2).count(k) for k in set(y_test_c2)}
```

Out[61]:
```
(1360,)
{0: 346, 1: 47, 2: 168, 3: 140, 4: 178, 5: 262, 6: 219}
```

# Model Building

## Logestic Regression

Need to select parameters as multinormial or OVR(one verses Rest)

In [62]:
```
lr_model_c2 = LogisticRegression(multi_class='multinomial')
lr_model_c2.fit(x_train_c2,y_train_c2)
```

Out[62]:
```
LogisticRegression(multi_class='multinomial')
```

## Random Forest

Compare LR model with a none Linear Model

In [63]:
```
rf_classifier_c2 = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 42)
rf_classifier_c2.fit(x_train_c2,y_train_c2)
```

Out[63]:
```
RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=42)
```

## K fold Cross Validation

In [89]:
```
kfold_validation=KFold(40)
results_kfold_lr_c2 = cross_val_score(lr_model_c2,X,y,cv=kfold_validation)
results_kfold_rf_c2 = cross_val_score(rf_classifier_c2,X,y,cv=kfold_validation)
print("K_Fold validation Score Logestic Regression(Linear Model) CASE2(90:10): ",np.mean(results_kfold_lr_c2))
print("K_Fold validation Score Random Forest(Linear Model) CASE2(90:10): ",np.mean(results_kfold_rf_c2))
```

```
K_Fold validation Score Logestic Regression(Linear Model) CASE2(90:10):  0.8070338799236509
K_Fold validation Score Random Forest(Linear Model) CASE2(90:10):  0.7896800711435017
```

After the kfold Validation we can see that the model is well build and generalized. We can safely state that the model built is free from Overfitting and uderfitting

## MLE in Linear regression

Let's consider linear regression. Imagine that each single prediction y^ produces a "conditional" distribution pmodel(y^ Ix), given a sufficiently large train set. The goal of the learning algorithm is again to match the distribution pdata(yIx). Now we need an assumption. We hypothesize the neural network or any estimator f as y^=f(x,θ). The estimator approximates the mean of the normal distribution N(μ,σ) the we choose to parametrize pdata. Specifically, in the simplest case of linear regression we have μ=θTx. We also assume a fixed standard deviation σ of the normal distribution. These assumptions immediately causes MLE to become Mean Squared Error (MSE) optimization. Let's see how.

$$\hat{y} = f(\mathbf{x}, \boldsymbol{\theta})$$
$$y \sim \mathcal{N}\left(y, \mu = \hat{y}, \sigma^2\right)$$
$$p(y \mid \mathbf{x}, \boldsymbol{\theta}) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(y-\hat{y})^2}{2\sigma^2}\right)$$

In terms of log-likelihood we can form a loss function:

$$
\begin{aligned}
L &= \sum_{i=1}^{m} \log p(y \mid \mathbf{x}, \boldsymbol{\theta}) \\
&= \sum_{i=1}^{m} \log \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-\left(\hat{y}^{(i)} - y^{(i)}\right)^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{m} -\log(\sigma\sqrt{2\pi}) - \log\exp\left(\frac{(\hat{y}^{(i)} - y^{(i)})^2.}{2\sigma^2}\right) \\
&= \sum_{i=1}^{m} -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{(\hat{y}^{(i)} - y^{(i)})^2}{2\sigma^2} \\
&= -m\log(\sigma) - \frac{m}{2}\log(2\pi) - \sum_{i=1}^{m} \frac{\left(\hat{y}^{(i)} - y^{(i)}\right)^2}{2\sigma^2}
\end{aligned}
$$

By taking the partial derivative with respect to the parameters, we get the desired MSE.

$$
\begin{aligned}
\nabla_\theta L &= -\nabla_\theta \sum_{i=1}^{m} \frac{\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2}{2\sigma^2} \\
&= -m\log(\sigma) - \frac{m}{2}\log(2\pi) - \sum_{i=1}^{m} \frac{\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2}{2\sigma^2} \\
&= -m\log(\sigma) - \frac{m}{2}\log(2\pi) - \frac{m}{2\sigma^2} MSE
\end{aligned}
$$

## MLE in supervised classification

In linear regression, we pmodel(y|x,θ) as a normal distribution. More precisely, we parametrized the mean to be μ=θTx. It is possible to convert linear regression to a classification problem. All we need to do is encode the ground truth as a one-hot vector:

$$
\begin{cases}
1 & \text{if } y = y_i \\
0 & \text{otherwise}
\end{cases}
$$

, where ii refer to a single data instance.

$$
\begin{aligned}
H_i\left(p_{data}, p_{model}\right) &= -\sum_{y \in Y} p_{data}\left(y \mid \mathbf{x}_i\right) \log p_{model}\left(y \mid \mathbf{x}_i\right) \\
&= -\log p_{model}\left(y_i \mid \mathbf{x}_i\right)
\end{aligned}
$$

For simplicity let's consider the binary case of two labels, 0 and 1.

$$
\begin{aligned}
L &= \sum_{i=1}^{n} H_i\left(p_{data}, p_{model}\right) \\
&= \sum_{i=1}^{n} -\log p_{model}\left(y_i \mid \mathbf{x}_i\right) \\
&= -\sum_{i=1}^{n} \log p_{model}\left(y_i \mid \mathbf{x}_i\right)
\end{aligned}
$$

$$
= \arg\min_{\boldsymbol{\theta}} L = \arg\min_{\boldsymbol{\theta}} -\sum_{i=1}^{n} \log p_{model}\left(y_i \mid \mathbf{x}_i\right)
$$

This is in line with our definition of conditional MLE:

θML=θargmaxi=1∑mlogpmodel(y(i)∣x(i),θ)

Broadly speaking, MLE can be applied to most (supervised) learning problems, by specifying a parametric family of (conditional) probability distributions. Another way to achieve this in a binary classification problem would be to take the scalar output y of the linear layer and pass it from a sigmoid function. The output will be in the range [0,1] and we define this as the probability of p(y=1∣x,θ).

p(y=1∣x,θ)=σ(θTx)=sigmoid(θTx)∈[0,1]

Consequently, p(y=0∣x,θ)=1−p(y=1∣x,θ). In this case binary-cross entropy is practically used. No closed form solution exist here, one can approximate it with gradient descend. For reference, this approach is surprisingly known as ""logistic regression".

## Regularization

Regularization helps us control our model capacity, ensuring that our models are better at making (correct) classifications on data points that they were not trained on, which we call the ability to generalize. If we don’t apply regularization, our classifiers can easily become too complex

and overfit to our training data, in which case we lose the ability to generalize to our testing data (and data points outside the testing set as well, such as new images in the wild).

However, too much regularization can be a bad thing. We can run the risk of underfitting, in which case our model performs poorly on the training data and is not able to model the relationship between the input data and output class labels (because we limited model capacity too much).

### Regularization for Case 1: 80:20

In [64]:
```
reg_case1 = []
# loop over our set of regularizers
for r in (None, "l1", "l2"):
        # train a SGD classifier using a softmax loss function and the
        # specified regularization function for 10 epochs
        print("[INFO] training model with '{}' penalty".format(r))
        model = SGDClassifier(loss="log", penalty=r, max_iter=100,
                learning_rate="constant", tol=1e-3, eta0=0.01, random_state=10)
        model.fit(x_train_c1, y_train_c1)
        # evaluate the classifier
        acc = model.score(x_test_c1, y_test_c1)
        reg_case1.append(acc)
        print("[INFO] {} penalty accuracy: {:.2f}%".format(r,
                acc * 100))
```

```
[INFO] training model with 'None' penalty
[INFO] None penalty accuracy: 84.85%
[INFO] training model with 'l1' penalty
[INFO] l1 penalty accuracy: 84.88%
[INFO] training model with 'l2' penalty
[INFO] l2 penalty accuracy: 84.81%
```

### Regularization for Case 2: 90:10

In [65]:
```
# loop over our set of regularizers
reg_case2 = []
for r in (None, "l1", "l2"):
        # train a SGD classifier using a softmax loss function and the
        # specified regularization function for 10 epochs
        print("[INFO] training model with '{}' penalty".format(r))
        model = SGDClassifier(loss="log", penalty=r, max_iter=100,
```

```
                learning_rate="constant", tol=1e-3, eta0=0.01, random_state=10)
        model.fit(x_train_c2, y_train_c2)
        # evaluate the classifier
        acc = model.score(x_test_c2, y_test_c2)
        reg_case2.append(acc)
        print("[INFO] {} penalty accuracy: {:.2f}%".format(r,
                acc * 100))
```

```
[INFO] training model with 'None' penalty
[INFO] None penalty accuracy: 85.96%
[INFO] training model with 'l1' penalty
[INFO] l1 penalty accuracy: 86.03%
[INFO] training model with 'l2' penalty
[INFO] l2 penalty accuracy: 86.03%
```

In [66]:
```
index_reg = ["Normal","L1","L2(Regularization(Weight Decay)"]
out_reg = pd.DataFrame(list(zip(reg_case1,reg_case2)),columns=[ "Case1","Case2"],index = [index_reg])
```

**Comparison for Regularization Output - Case 1 & 2**

In [67]:
```
out_reg
```

Out[67]:

| | Case1 | Case2 |
|---|---|---|
| **Normal** | 0.848474 | 0.859559 |
| **L1** | 0.848841 | 0.860294 |
| **L2(Regularization(Weight Decay)** | 0.848106 | 0.860294 |

Since, our model is neither underfit or overfit, the outputs for both case 1 as well as 2 even post regulrization is comparable and close to the output without regularization. However, performance with 90:10 split is slightly better than 80:20 split.

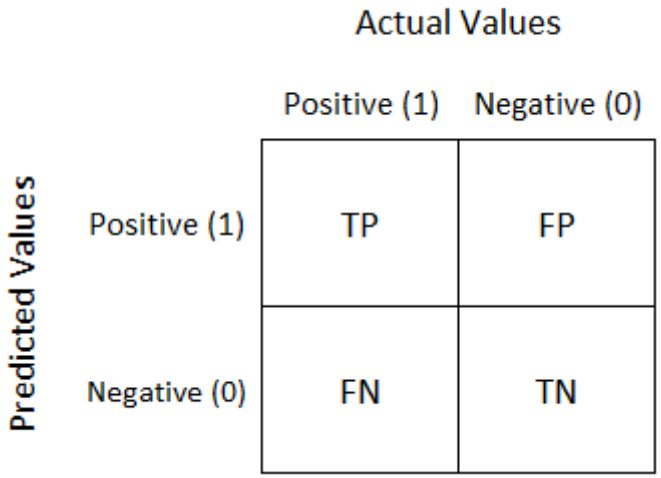
## Performance Evaluation

The Logestic Regression Models build give the following predictions and are used for evaluation of the models

In [68]:
```
y_pred_c1 = lr_model_c1.predict(x_test_c1)
y_pred_c2 = lr_model_c2.predict(x_test_c2)
```

## Confusion Matrix

performance measurement for machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values.




It is extremely useful for measuring Recall, Precision, Specificity, Accuracy, and most importantly AUC-ROC curves.

In [71]:
```
dict_Class
```

Out[71]:
```
{'DERMASON': 0,
 'BOMBAY': 1,
 'CALI': 2,
 'BARBUNYA': 3,
 'HOROZ': 4,
 'SIRA': 5,
 'SEKER': 6}
```

In [72]:
```
print("Confusion Matrix from Case1(80:20)")
confusion_matrix(y_test_c1,y_pred_c1)
```

```
Confusion Matrix from Case1(80:20)
```

Out[72]:
```
array([[632,   0,   0,   0,   2,  46,  20],
       [  0,   7,  13,   6,   4,  57,   0],
       [  0,   2, 292,  35,   7,   2,   1],
       [  0,   1,  33, 210,   4,   5,   1],
       [  1,   0,   6,   1, 358,   7,   0],
       [ 40,   9,   3,   3,  13, 457,  15],
       [ 21,   2,   0,   0,   0,  14, 389]], dtype=int64)
```

In [73]:
```
print("Confusion Matrix from Case1(90:10)")
confusion_matrix(y_test_c2,y_pred_c2)
```

Out[73]:
```
Confusion Matrix from Case1(90:10)
array([[315,   0,   0,   0,   1,  20,  10],
       [  0,   4,   8,   5,   1,  29,   0],
       [  0,   0, 150,  15,   2,   0,   1],
       [  0,   1,  18, 115,   3,   3,   0],
       [  0,   0,   2,   0, 174,   2,   0],
       [ 13,   4,   2,   1,  10, 227,   5],
       [ 10,   1,   0,   0,   0,   7, 201]], dtype=int64)
```

## Accuracy,Precision, Recall, f1 score,

**Accuracy :** The base metric used for model evaluation is often Accuracy, describing the number of correct predictions over all predictions

**Accuracy = (TP + TN)/(TP + FP + TN + FN)**

**Recall** : Recall is a measure of how many of the positive cases the classifier correctly predicted, over all the positive cases in the data

**Recall = TruePositives / (TruePositives + FalseNegatives)**

**Precision** : Precision is a measure of how many of the positive predictions made are correct (true positives)

**Precision = TruePositives / (TruePositives + FalsePositives)**

**F1 Score** :F1-Score is a measure combining both precision and recall. It is generally described as the harmonic mean of the two.

**F-Measure = (2 * Precision * Recall) / (Precision + Recall)**

In [74]:
```
matrix_c1 = [
    accuracy_score(y_test_c1,y_pred_c1),
    precision_score(y_test_c1,y_pred_c1,average="weighted",),
    recall_score(y_test_c1,y_pred_c1,average="weighted"),
```

```
    f1_score(y_test_c1,y_pred_c1,average="weighted")
]
```

In [75]:
```
matrix_c2 = [
    accuracy_score(y_test_c2,y_pred_c2),
    precision_score(y_test_c2,y_pred_c2,average="weighted"),
    recall_score(y_test_c2,y_pred_c2,average="weighted"),
    f1_score(y_test_c2,y_pred_c2,average="weighted")
]
```

### Final Comparison for Case 1 and Case 2

In [76]:
```
index_matrix = ["Accuracy","Precision","Recall","F1-Score"]
out_matrix = pd.DataFrame(list(zip(matrix_c1,matrix_c2)),columns=[ "Case1","Case2"],index = [index_matrix])
out_matrix
```

Out[76]:

| | Case1 | Case2 |
|---|---|---|
| **Accuracy** | 0.862449 | 0.872059 |
| **Precision** | 0.850961 | 0.861143 |
| **Recall** | 0.862449 | 0.872059 |
| **F1-Score** | 0.853872 | 0.862379 |

The comparison for accuracy, precision, recall and f1-score shows that data is comparable and the model has shown good performance. We cannot alone rely on accuracy score as it will be biased, so we also calculate precision (where false positives hold imporatance) and recall scores (where false negatives hold imporatance) and ultimately find F1-score (which takes both precision and recall into account) in order to do effective performance evaluation.

# Conclusion

We performed multi-class classification on dry beans dataset. To achieve the same data below steps were followed -

1. Exploratory Data Analysis -
2. Feature Engineering
3. Feature Selection

4. Model Building -
   i. Implemented using Logistic Regression (Linear) and Random Forest Classifier (Non-Linear)
   ii. Implemented for two scenarios - data split into training and test as per 80:20 and 90:10 split
5. Performance Evaluation -
   i. Performance evaluation done using confusion matrix, K-fold validation, precision, recall, accuracy and F-score
   ii. Evaluation shows that performances for 90:10 and 80:20 splits are comparable however, it is slightly better for 90:10 split.
   iii. Linear model performs better in some scenarios since, it tends to do more generalization compared to non-linear models.

In [ ]: