# Assignment 1.

2020249.

## Section 1 theory.

Part a)

given $\bar{x}$ & $\bar{y}$ are arithmetic mean of x & y respectively

X -> independ variable.

Y -> Dependent variable.

$$\bar{y} = w_0 + w_1 x.$$

With noise. (generisation)

$$y_i = w_0 + w_i x_i + \varepsilon$$

n = number of Samples in data

$$\sum_{i=1}^{n} y_i = n(w_0 + w_i x_i + \varepsilon_i)$$

$$= nw_0 + w_i \sum_{i=1}^{n} x_i + \underbrace{\sum_{i=1}^{n} \varepsilon_i}_{\text{noise considered zero.}}$$

$$\frac{1}{n} \sum_{i=1}^{n} \frac{y_i}{\bar{y}} = w_0 + \frac{w_1}{n} \sum_{i=1}^{n} x_i$$

$$\bar{y} = w_0 + w_1 \bar{x}$$

$\sum_{i=1}^{n} \frac{y_i}{n}$ = arithmetic mean of $y$.

$\frac{1}{n} \sum_{i=1}^{n} x_i$ = arithmetic mean of $X$.

b) No, (Not always)

example.

let X, Y, Z are Random variables.

X & Y are positively correlated ∵
Y & Z are positively correlated.

~~$C(X,Y) \geq C(Y,Z) * C(Z,X) - \sqrt{1 - C(Y,Z)^2 \times (1}$~~

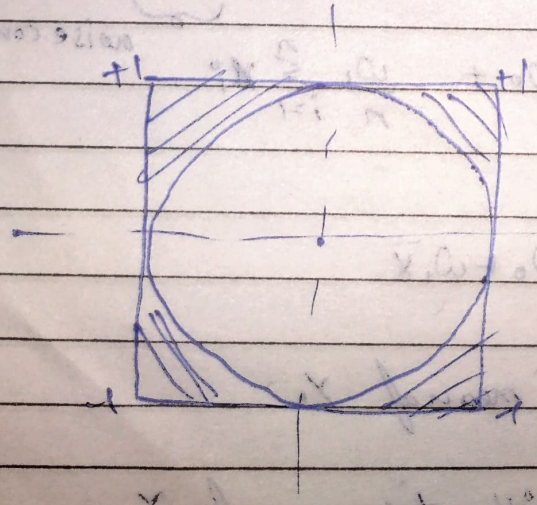
$$C(X,Y) \geq C(Y,Z) * C(Z,X) - \sqrt{(1 - C(Y,Z)^2) \times (1 - C(Z,X)^2)}$$

for $C(X,Y)$ to be more than zero.

$$C(Y,Z) \times C(Z,X) > \sqrt{(1 - C(Y,Z)^2)(1 - C(Z,X)^2)}$$

Squaring both Sides.

$$C(Y,Z)^2 + C(Z,X)^2 > 1$$




if two correlations are in the Circular Non-shaded region we can't say precisely anything about the Correlation of the 3rd pair.

## Proof of LLN (WLLN)

Let $X_1 \cdots X_n$ are ~~Dati~~ i.i.d Random variables with mean give $\mu_x$ where n is very large.

$$S_n = \frac{X_1 + X_2 \cdots X_n}{n}$$

$$\lim_{n \to \infty} P\{|S_n - \mu_x| > \varepsilon\} = 0.$$

Also.

$E[S_n] = \mu_x$ and variance $= \frac{n\sigma^2}{n^2} = \frac{\sigma^2}{n}$.

From chebyshev's inequality

$$P\{|S_n - \mu_x| > \varepsilon\} \leq \frac{Var[S_n]}{\varepsilon^2} = \frac{\sigma_x^2}{n\varepsilon^2}$$

$$\lim_{n \to \infty} P\{|S_n - \mu_x| \geq \varepsilon\} = 0$$

Example.

Let's consider a fair die distribution.

$$S = \{1, 2, 3, 4, 5, 6\}.$$

$$S_n = 3.5.$$

When we roll the dice for a very large number(n) times.

the average value approaches 3.5

Pseudo code

```
def FLLN(die, n)
    result = []
    for i in Range (1, n+1)
        result.append (random.choice(1, 2, 3, 4, 5, 6))
    return result

def LLN(n)
    result = LLPDie(n)
    average = []
    for i in Range (len(Results)):
        average.append (np.cumsum[i]/(i+1))

    return average.
```

Q.4 MAP Solution for linear regression.

from Bayes th.

$$P(w|D) = \frac{P(D|w) \times P(w)}{P(D)} \leftarrow \text{unknown.}$$

$$P(w|D) \propto P(D|w) \times P(w)$$

take log

$$\log(P(w|D)) = \log P(D|w) + \log(P(w)) \quad \text{---(1)}$$

assume a gaussian distribution for $w$ with mean 0 and variance $\sigma$, $N(0, \sigma^2)$

$$\log(P|w) = \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(\frac{-\sigma}{2} w^T w\right)}\right) = \log(P|w)$$

$$= -\left(\frac{1}{2\sigma^2} w^T w\right)$$

$$\log(P(D|w)) = \frac{1}{2\sigma^2} (y - Dw)^T (y - Dw)$$

put $P(D|w)$ & $P(w)$ in eq (1) & differentiate w.r.t $w$.

$$\frac{d}{dw}\left[\frac{1}{2\sigma^2}(y-Dw)^T(y-Dw)\right] + \frac{d}{dw}\left(\frac{1}{2\sigma^2}w^Tw\right)$$

$$= \frac{1}{\sigma^2}\left(w^TD^TD - y^TD\right) + \frac{1}{\sigma^2}w^T$$

$$\frac{1}{\sigma^2}\left(w^TD^TD - y^TD\right) + \frac{1}{\sigma^2}w^T = 0$$

$$\frac{w^T}{\sigma^2}\left(D^TD + I\right) = \frac{y^TD}{\sigma^2}$$

$$w^T = (y^TD)(D^TD + I)^{-1}$$

$$w^T_{map} = y^TD\,(D^TD + \lambda I)^{-1}$$

↖ regulizer.

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
```

In [2]:
```
df=pd.read_csv("Real estate.csv",index_col=False)
```

In [3]:
```
df=df.drop("No",axis=1)
df
```

Out[3]:

| | X1 transaction date | X2 house age | X3 distance to the nearest MRT station | X4 number of convenience stores | X5 latitude | X6 longitude | Y house price of unit area |
|---|---|---|---|---|---|---|---|
| **0** | 2012.917 | 32.0 | 84.87882 | 10 | 24.98298 | 121.54024 | 37.9 |
| **1** | 2012.917 | 19.5 | 306.59470 | 9 | 24.98034 | 121.53951 | 42.2 |
| **2** | 2013.583 | 13.3 | 561.98450 | 5 | 24.98746 | 121.54391 | 47.3 |
| **3** | 2013.500 | 13.3 | 561.98450 | 5 | 24.98746 | 121.54391 | 54.8 |
| **4** | 2012.833 | 5.0 | 390.56840 | 5 | 24.97937 | 121.54245 | 43.1 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **409** | 2013.000 | 13.7 | 4082.01500 | 0 | 24.94155 | 121.50381 | 15.4 |
| **410** | 2012.667 | 5.6 | 90.45606 | 9 | 24.97433 | 121.54310 | 50.0 |
| **411** | 2013.250 | 18.8 | 390.96960 | 7 | 24.97923 | 121.53986 | 40.6 |
| **412** | 2013.000 | 8.1 | 104.81010 | 5 | 24.96674 | 121.54067 | 52.5 |
| **413** | 2013.500 | 6.5 | 90.45606 | 9 | 24.97433 | 121.54310 | 63.9 |

414 rows × 7 columns

In [4]:
```
for i in df.columns[:-1]:
    print(i)
```

```
X1 transaction date
X2 house age
X3 distance to the nearest MRT station
X4 number of convenience stores
X5 latitude
X6 longitude
```

In [5]:
```
df.describe()
```

Out[5]:

| | X1 transaction date | X2 house age | X3 distance to the nearest MRT station | X4 number of convenience stores | X5 latitude | X6 longitude | Y house price of unit area |
|---|---|---|---|---|---|---|---|
| **count** | 414.000000 | 414.000000 | 414.000000 | 414.000000 | 414.000000 | 414.000000 | 414.000000 |

| | X1 transaction date | X2 house age | X3 distance to the nearest MRT station | X4 number of convenience stores | X5 latitude | X6 longitude | Y house price of unit area |
|---|---|---|---|---|---|---|---|
| **mean** | 2013.148971 | 17.712560 | 1083.885689 | 4.094203 | 24.969030 | 121.533361 | 37.980193 |
| **std** | 0.281967 | 11.392485 | 1262.109595 | 2.945562 | 0.012410 | 0.015347 | 13.606488 |
| **min** | 2012.667000 | 0.000000 | 23.382840 | 0.000000 | 24.932070 | 121.473530 | 7.600000 |
| **25%** | 2012.917000 | 9.025000 | 289.324800 | 1.000000 | 24.963000 | 121.528085 | 27.700000 |
| **50%** | 2013.167000 | 16.100000 | 492.231300 | 4.000000 | 24.971100 | 121.538630 | 38.450000 |
| **75%** | 2013.417000 | 28.150000 | 1454.279000 | 6.000000 | 24.977455 | 121.543305 | 46.600000 |
| **max** | 2013.583000 | 43.800000 | 6488.021000 | 10.000000 | 25.014590 | 121.566270 | 117.500000 |

In [6]:
```
df1=pd.read_csv("Real estate.csv")
```

In [7]:
```
class Linear_Regression():
    def __init__(self,l_rate,Epochs):

        self.l_rate=l_rate
        self.Epochs=Epochs

    def fit(self, X, Y ):

        self.m,self.n = X.shape
        on=np.ones((self.m,1))
        self.X=np.concatenate((on,X),axis=1)
        self.n=self.n+1
        self.Y=Y
        self.W=np.zeros(self.n)
        self.H=np.dot(self.X,self.W)
        self.Cost=np.zeros(self.Epochs)

        for i in range(self.Epochs):
            self.upd_weight()
            self.Cost[i]=np.sum(np.square(self.H-self.Y))/(2*self.m)
        return self

    def upd_weight(self):

        self.W[0]=self.W[0]-(self.l_rate/self.m)*sum(self.H-self.Y)
        for i in range(1,self.n):
            self.W[i]=self.W[i]-(self.l_rate/self.m)*sum((self.H-self.Y)*self.X[:,i])
        self.H=np.dot(self.X,self.W)
        return self

    def W_C(self):

        print(self.W[0])
        print(self.W[1:])

    def predict(self,X):

        return X.dot(self.W[1:])+self.W[0]
```

```
def norm(df1):
    for i in df1.columns[:]:
        df1[i]=df1[i]-df1[i].mean()
        df1[i]=df1[i]/np.sqrt(df1[i].var())
    #return np.array(df1)
    return df1
```

## Part a

You will need to perform K-Fold cross-validation (K=2-5) in this exercise (implement from scratch). What is the optimal value of K? Justify it in your report along with the table for the mean accuracy of K-values and K-value.

In [8]:

```
def K_fold_validation(k,df):
    n=len(df.index)
    fold_size=n//k
    if(n%k>0):
        fold_size+=1
    df = df.reindex(np.random.permutation(df.index))
    df = df.reset_index(drop=True)
    df=np.array(df)
    #print(fold_size)
    l=[]
    k=0
    while(k<n):
        l.append(df[k:k+fold_size,:])
        k+=fold_size


#         print("=============================================================================")

    X_Trains=[]
    Y_Trains=[]
    X_Tests=[]
    Y_Tests=[]
    for i in range(len(l)):
        temp=[]
        for j in range(len(l)):
            if(j!=i):
                temp.append(pd.DataFrame(l[j]))
        t_df=np.array(pd.concat(temp))
        X_Trains.append(t_df[:,:-1])
        Y_Trains.append(t_df[:,-1])
        X_Tests.append(l[i][:,:-1])
        Y_Tests.append(l[i][:,-1])

    return X_Trains,Y_Trains,X_Tests,Y_Tests
```

In [9]:

```
results_mse=[]
# results_mle=[]
for i in range(2,6):
    l=[]
#     l1=[]
```

```
    X_Trains, Y_Trains, X_Tests, Y_Tests = K_fold_validation(i,norm(pd.read_csv("Real e
    for j in range(i):
        model=Linear_Regression(0.01,20000)
        model.fit(X_Trains[j],Y_Trains[j])
        y_pred=model.predict(X_Tests[j])
        difference_array = np.subtract(y_pred, Y_Tests[j])
        squared_array = np.square(difference_array)
        mse = squared_array.mean()
        l.append(mse)
#         l1.append(difference_array.mean())
    results_mse.append(np.sum(l)/len(l))
#     results_mle.append(np.sum(l1)/len(l1))
```

In [10]:

```
print("K| MSE ")
for i in range(2,6):
    print("{}| {}".format(i,results_mse[i-2]))
```

```
K| MSE
2| 0.48138327722866797
3| 0.43327257038518413
4| 0.42797314694324884
5| 0.4320416914330435
```

If we look at Mse k=5 is the best

## Part B

Plot the RMSE V/s iteration graph for all models trained with optimal value of K for K-Fold cross-validation. RMSE should be reported on the train and val set.

Choosen K = 5

In [11]:

```
#added the RMSE list for validation as well tranning set to the model

class Linear_Regression1():
    def __init__(self,l_rate,Epochs):

        self.l_rate=l_rate
        self.Epochs=Epochs

    def fit(self, X, Y ,X_test,Y_test):

        self.m,self.n = X.shape
        on=np.ones((self.m,1))
        self.X=np.concatenate((on,X),axis=1)
        self.n=self.n+1
        self.Y=Y
        self.W=np.zeros(self.n)
        self.H=np.dot(self.X,self.W)
        self.Cost=np.zeros(self.Epochs)
        self.val_err=[]
        self.tr_err=[]
```

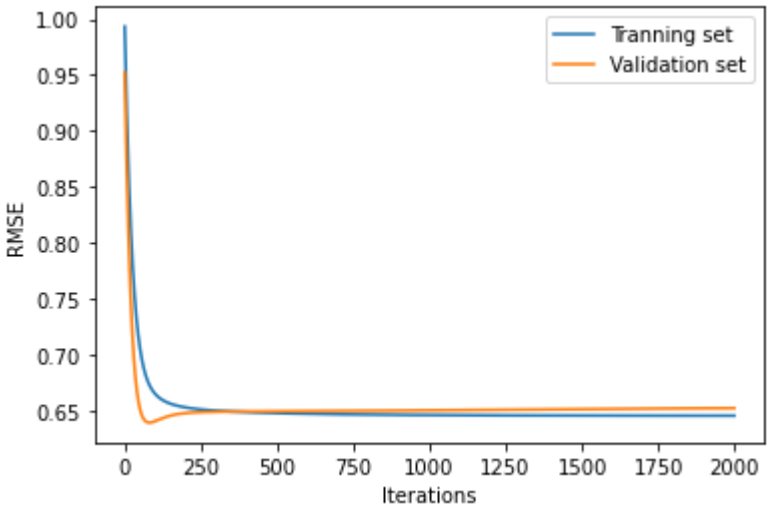
```
        for i in range(self.Epochs):
            self.upd_weight()
            self.val_err.append(self.rmse(Y_test,self.predict(X_test)))
            self.tr_err.append(self.rmse(Y,self.predict(X)))
            self.Cost[i]=np.sum(np.square(self.H-self.Y))/(2*self.m)
        #self.plot1()
        print("RMSE for tranning set => {}".format(self.tr_err[-1]))
        print("RMSE for validation set => {}".format(self.val_err[-1]))
        return self

    def upd_weight(self):

        self.W[0]=self.W[0]-(self.l_rate/self.m)*sum(self.H-self.Y)
        for i in range(1,self.n):
            self.W[i]=self.W[i]-(self.l_rate/self.m)*sum((self.H-self.Y)*self.X[:,i])
        self.H=np.dot(self.X,self.W)
        return self

    def W_C(self):
        print(self.W[0])
        print(self.W[1:])

    def rmse(self,y1,y2):
        mse = (np.square(y1 - y2)).mean()
        return np.sqrt(mse)
    def plot1(self):
        list1 = list(range(0,self.Epochs))

        plt.plot(list1,self.tr_err,label='Tranning set')
        plt.plot(list1,self.val_err,label='Validation set')
        plt.xlabel('Iterations')
        plt.ylabel('RMSE')
#         plt.title("k= "+str(i)+" and model number="+str(j+1)+"th")
        plt.legend()
        plt.show()

        plt.plot(list1,self.Cost,color='g',label='cost')
#         plt.title("Cost function for K = "+str(i)+" and model number="+str(j+1))
        plt.xlabel('Iterations')
        plt.ylabel('Cost')
        plt.legend()
        plt.show()
    def predict(self,X):

        return X.dot(self.W[1:])+self.W[0]
```

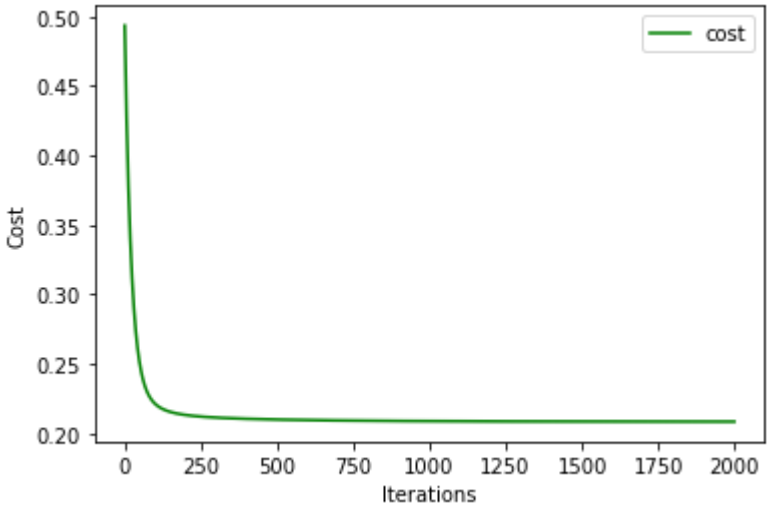
In [12]:

```
# the range for which one needs to run K-fold (K=5 ) so range (5,6)
for i in range(5,6):
    l=[]
    l1=[]
    X_Trains, Y_Trains, X_Tests, Y_Tests = K_fold_validation(i,norm(pd.read_csv("Real e
    for j in range(i):
        model=Linear_Regression1(0.01,2000)
        print("k= "+str(i)+" and model number="+str(j+1))
        model.fit(X_Trains[j],Y_Trains[j],X_Tests[j],Y_Tests[j])
        model.plot1()
```

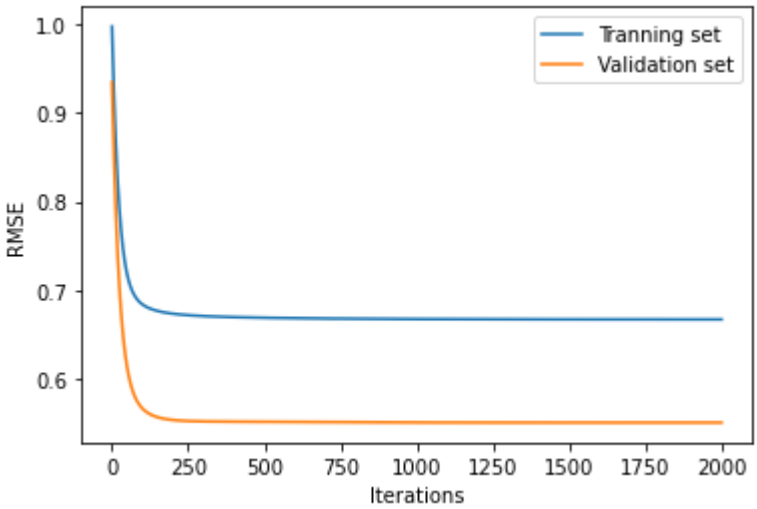
```
k= 5 and model number=1
```

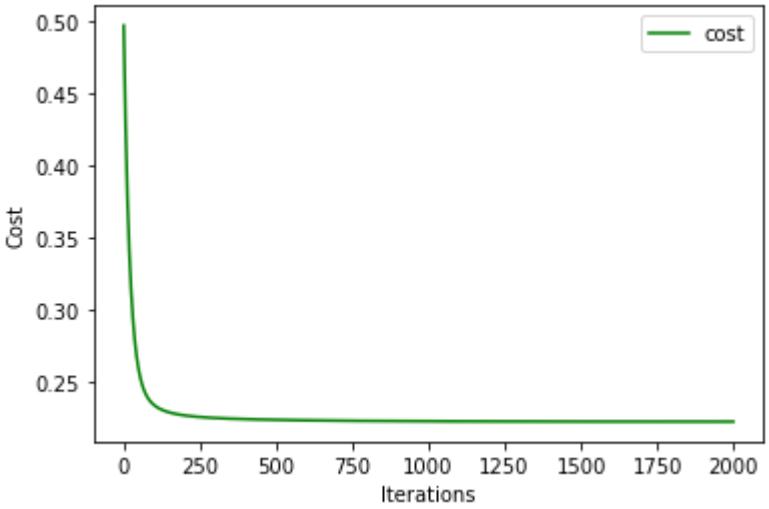
```
RMSE for tranning set => 0.6455807116763345
RMSE for validation set => 0.652294284531606
```



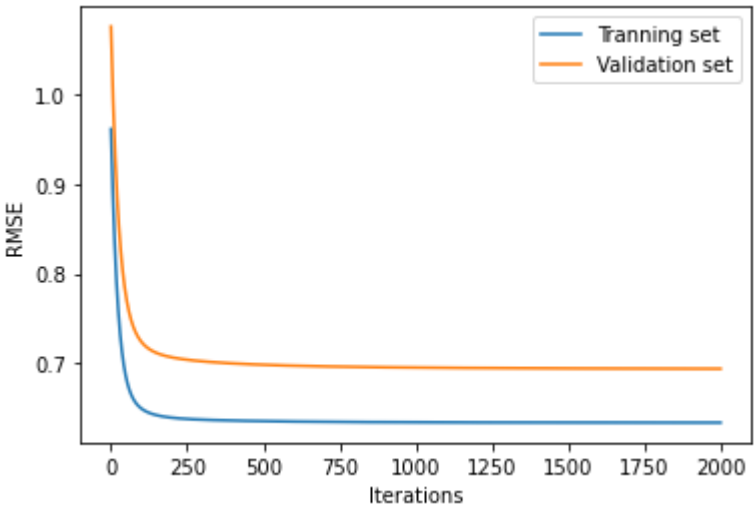



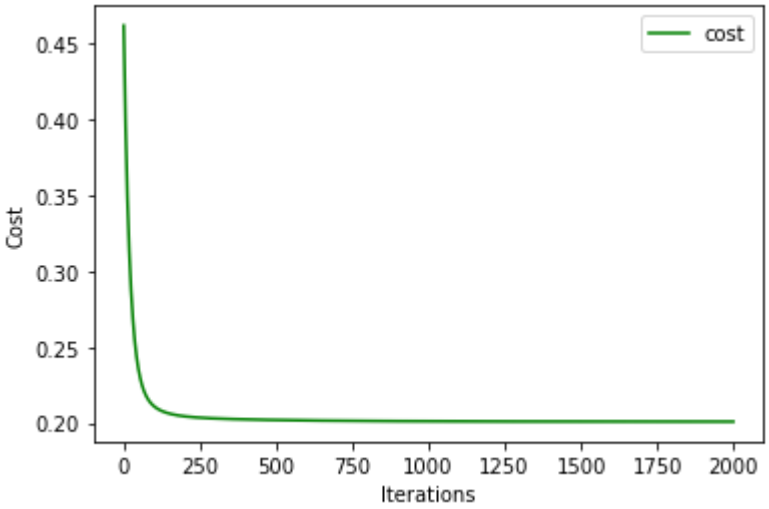

```
k= 5 and model number=2
RMSE for tranning set => 0.6673322954749914
RMSE for validation set => 0.5511216204356201
```



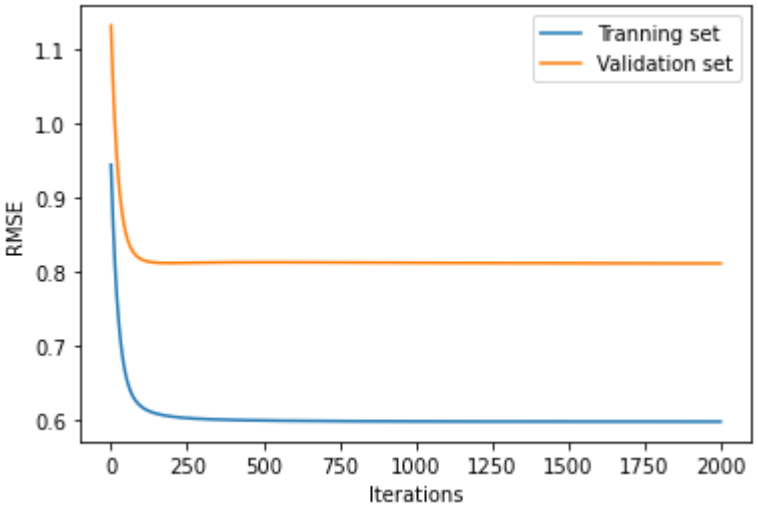


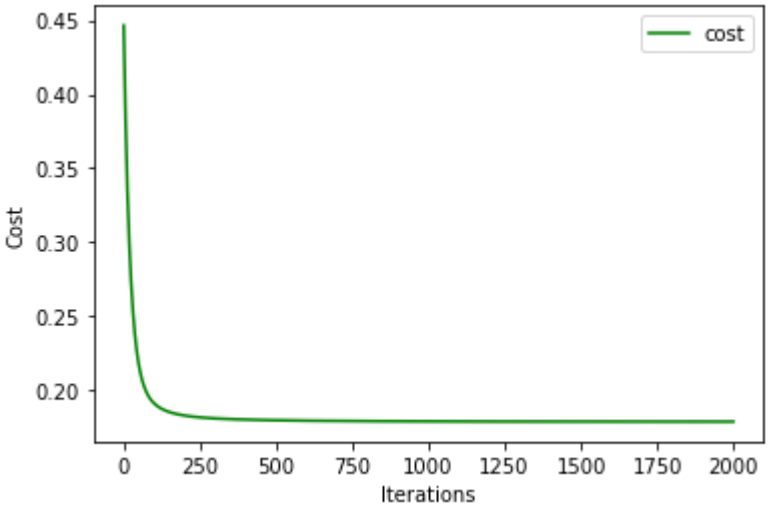

```
k= 5 and model number=3
RMSE for tranning set => 0.6337213018795999
RMSE for validation set => 0.6938312819257478
```



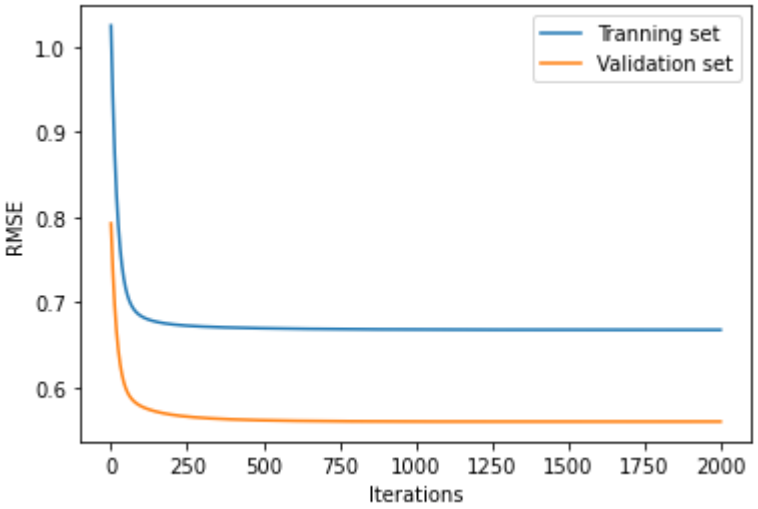



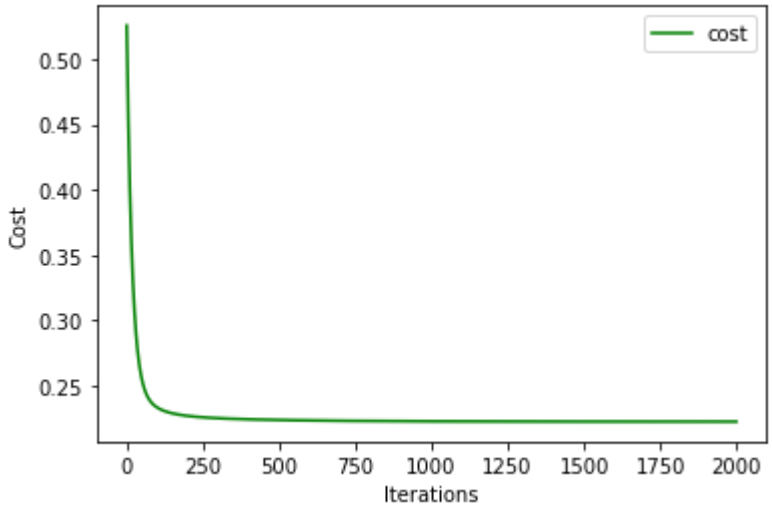

```
k= 5 and model number=4
RMSE for tranning set => 0.5978329170873334
RMSE for validation set => 0.8115815436018022
```




```
k= 5 and model number=5
RMSE for tranning set => 0.6673537073840736
RMSE for validation set => 0.5595258522593299
```






## Part C

Modify your Regression implementation by including L1 (LASSO) and L2 (Ridge Regression) regularization. Implement both regularization function from scratch and train the model again. Try different values of the regularization parameter and report the best one. Plot similar RMSE V/s iteration graph as before

In [13]:

```
class Linear_Regression_lasso():
    def __init__(self,l_rate,Epochs,lmd):

        self.l_rate=l_rate
        self.Epochs=Epochs
        self.lmd=lmd

    def fit(self, X, Y ,X_test,Y_test):

        self.m,self.n = X.shape
        on=np.ones((self.m,1))
        self.X=np.concatenate((on,X),axis=1)
        self.n=self.n+1
        self.Y=Y
        self.W=np.zeros(self.n)
        self.H=np.dot(self.X,self.W)
        self.Cost=np.zeros(self.Epochs)
        self.val_err=[]
        self.tr_err=[]



        for i in range(self.Epochs):
            self.upd_weight()
            self.val_err.append(self.rmse(Y_test,self.predict(X_test)))
            self.tr_err.append(self.rmse(Y,self.predict(X)))
            self.Cost[i]=np.sum(np.square(self.H-self.Y))/(2*self.m)
        #self.plot1()
        return self

    def upd_weight(self):
        p=0
```

```
        if(self.W[0]<0):
            p=1
        elif(self.W[0]>0):
            p=-1

        self.W[0]=self.W[0]-(self.l_rate/self.m)*sum(self.H-self.Y)+(p*self.lmd)
        for i in range(1,self.n):
            p=0
            if(self.W[0]<0):
                p=1
            elif(self.W[0]>0):
                p=-1
            self.W[i]=self.W[i]-(self.l_rate/self.m)*sum((self.H-self.Y)*self.X[:,i])+(

        self.H=np.dot(self.X,self.W)
        return self

    def W_C(self):
        print(self.W[0])
        print(self.W[1:])

    def rmse(self,y1,y2):
        mse = (np.square(y1 - y2)).mean()
        return np.sqrt(mse)
    def plot1(self):
        list1 = list(range(0,self.Epochs))

        plt.plot(list1,self.tr_err,label='Tranning set')
        plt.plot(list1,self.val_err,label='Validation set')
#         plt.title("k= "+str(i)+" and model number="+str(j+1)+"th")
        plt.legend()
        plt.xlabel('Iterations')
        plt.ylabel('RMSE')
        plt.show()

        plt.plot(list1,self.Cost,color='g',label='cost')
#         plt.title("Cost function for K = "+str(i)+" and model number="+str(j+1))
        plt.legend()
        plt.xlabel('Iterations')
        plt.ylabel('Cost')
        plt.show()

    def predict(self,X):

        return X.dot(self.W[1:])+self.W[0]
```

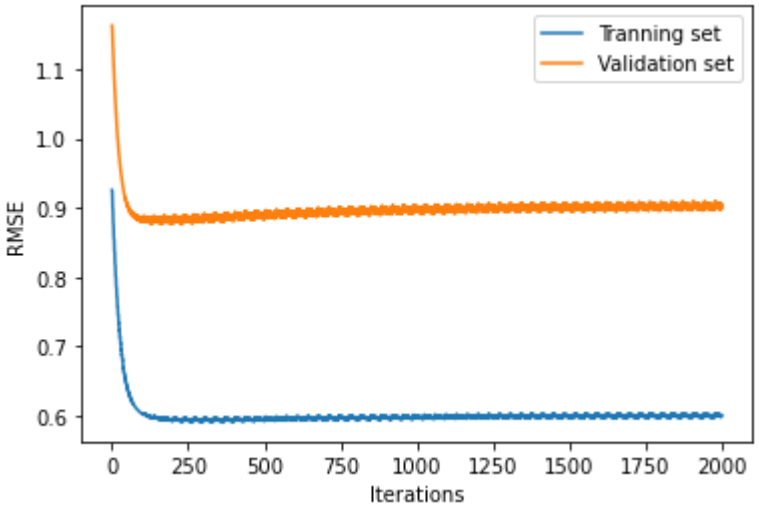
In [14]:

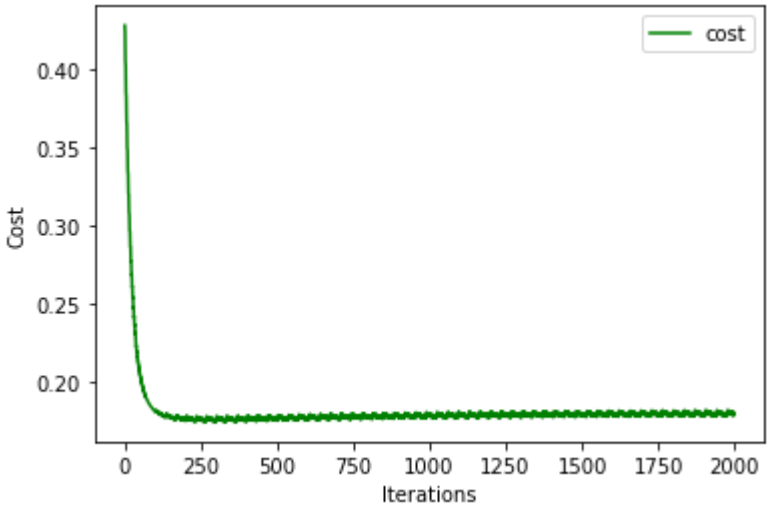
```
class Linear_Regression_ridge():
    def __init__(self,l_rate,Epochs,lmd):

        self.l_rate=l_rate
        self.Epochs=Epochs
        self.lmd=lmd

    def fit(self, X, Y ,X_test,Y_test):

        self.m,self.n = X.shape
        on=np.ones((self.m,1))
        self.X=np.concatenate((on,X),axis=1)
```

```
        self.n=self.n+1
        self.Y=Y
        self.W=np.zeros(self.n)
        self.H=np.dot(self.X,self.W)
        self.Cost=np.zeros(self.Epochs)
        self.val_err=[]
        self.tr_err=[]



        for i in range(self.Epochs):
            self.upd_weight()
            self.val_err.append(self.rmse(Y_test,self.predict(X_test)))
            self.tr_err.append(self.rmse(Y,self.predict(X)))
            self.Cost[i]=np.sum(np.square(self.H-self.Y))/(2*self.m)
        #self.plot1()
        return self

    def upd_weight(self):

        self.W[0]=((1-2*self.l_rate*self.lmd)*self.W[0])-(self.l_rate/self.m)*sum(self.

        for i in range(1,self.n):
            self.W[i]=((1-2*self.l_rate*self.lmd)*self.W[i])-(self.l_rate/self.m)*sum((

        self.H=np.dot(self.X,self.W)
        return self

    def W_C(self):
        print(self.W[0])
        print(self.W[1:])

    def rmse(self,y1,y2):
        mse = (np.square(y1 - y2)).mean()
        return np.sqrt(mse)

    def plot1(self):
        list1 = list(range(0,self.Epochs))

        plt.plot(list1,self.tr_err,label='Tranning set')
        plt.plot(list1,self.val_err,label='Validation set')
#         plt.title("k= "+str(i)+" and model number="+str(j+1)+"th")
        plt.legend()
        plt.xlabel('Iterations')
        plt.ylabel('RMSE')
        plt.show()

        plt.plot(list1,self.Cost,color='g',label='cost')
#         plt.title("Cost function for K = "+str(i)+" and model number="+str(j+1))
        plt.legend()
        plt.xlabel('Iterations')
        plt.ylabel('Cost')
        plt.show()

    def predict(self,X):

        return X.dot(self.W[1:])+self.W[0]
```

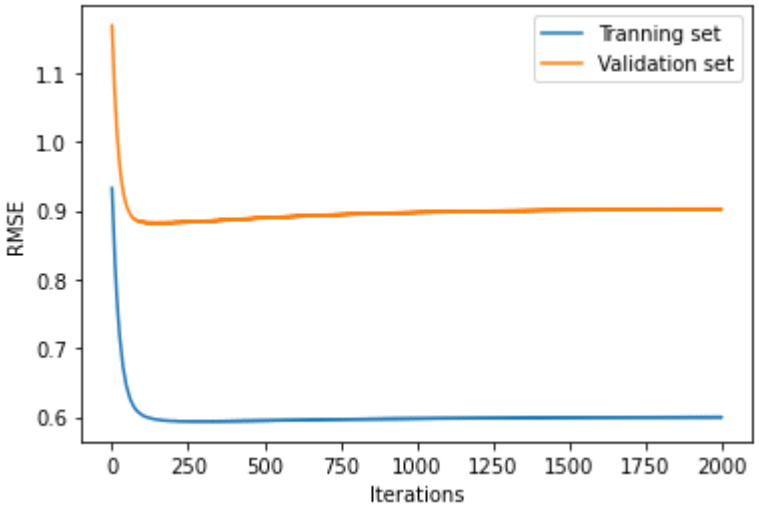
In [15]:

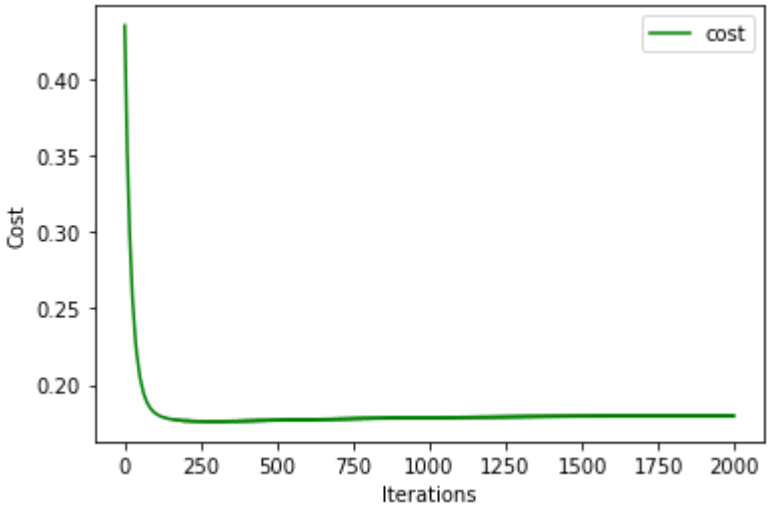
```
for i in range(5,6):
    l=[]
    l1=[]
    X_Trains, Y_Trains, X_Tests, Y_Tests = K_fold_validation(i,norm(pd.read_csv("Real e
    for j in range(i):
        mp=[0.01,0.001,0.000005,0.0006,0.00007]
        for m in range(5):
            model1=Linear_Regression_lasso(0.01,2000,mp[m])
            model1.fit(X_Trains[j],Y_Trains[j],X_Tests[j],Y_Tests[j])

            #we got the best split at k=5 and the 1th split so only showing that one
            if(j==1):
                model1.plot1()
                print("Regularisation parameter {}".format(mp[m]))
                print("Validation Set RMSE=> {}".format(model1.val_err[-1]))
                print("Tranning Set RMSE => {}".format(model1.tr_err[-1]))
```



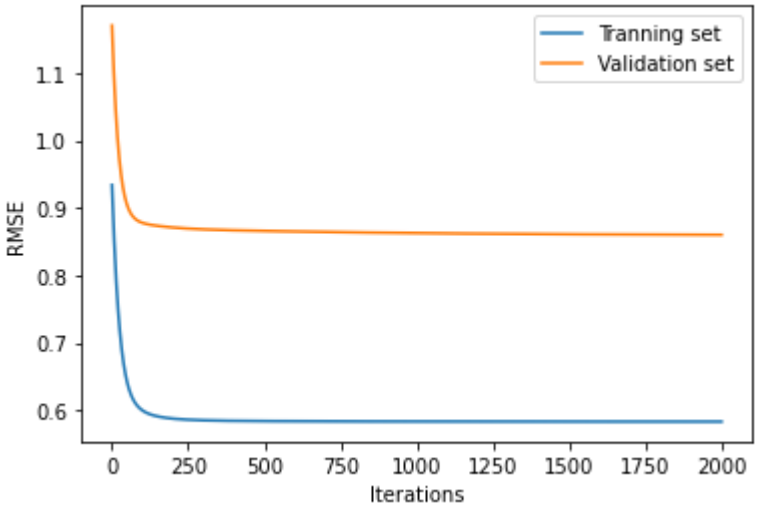



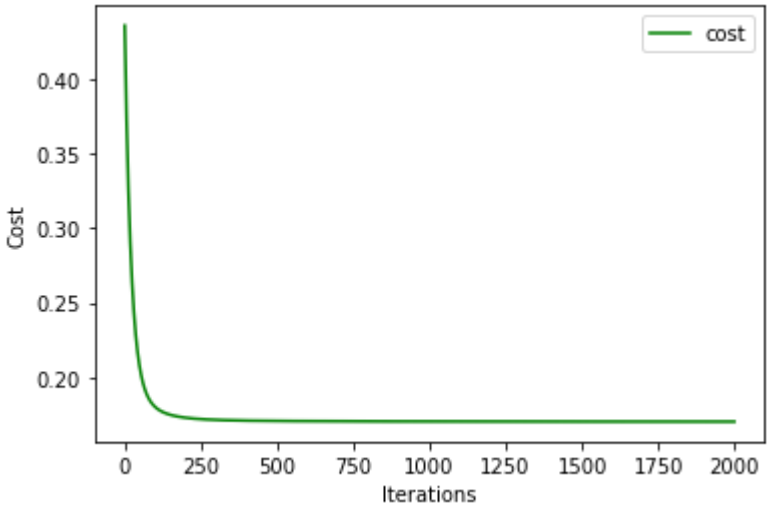

```
Regularisation parameter 0.01
Validation Set RMSE=> 0.9052595364454838
Tranning Set RMSE => 0.6004640526698816
```



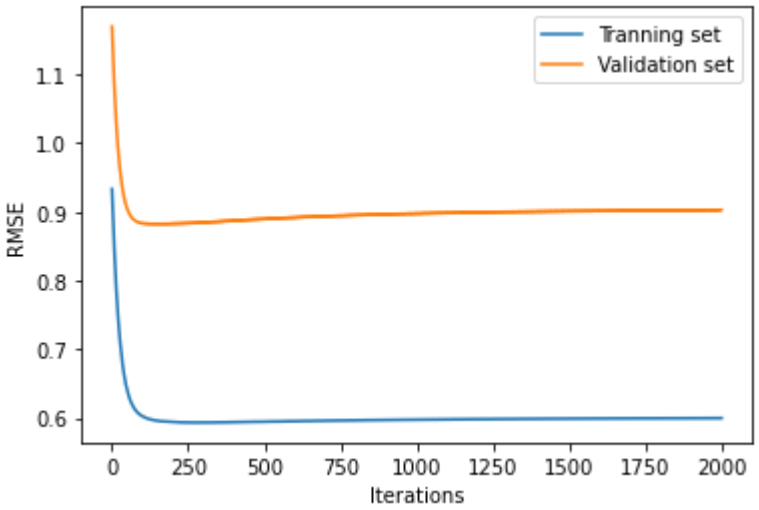



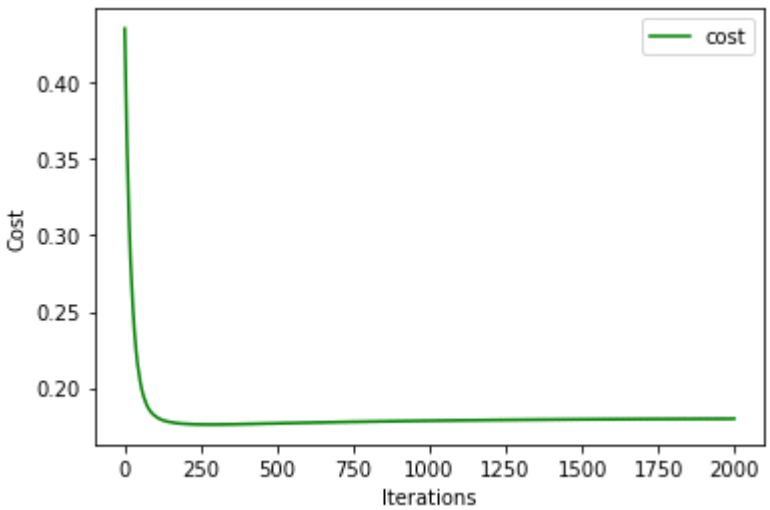

```
Regularisation parameter 0.001
Validation Set RMSE=> 0.9021533241674045
Tranning Set RMSE => 0.5996499679753795
```



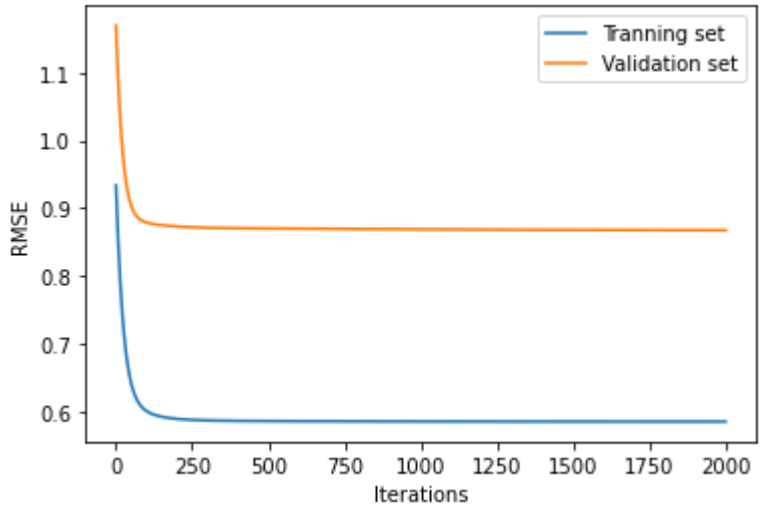


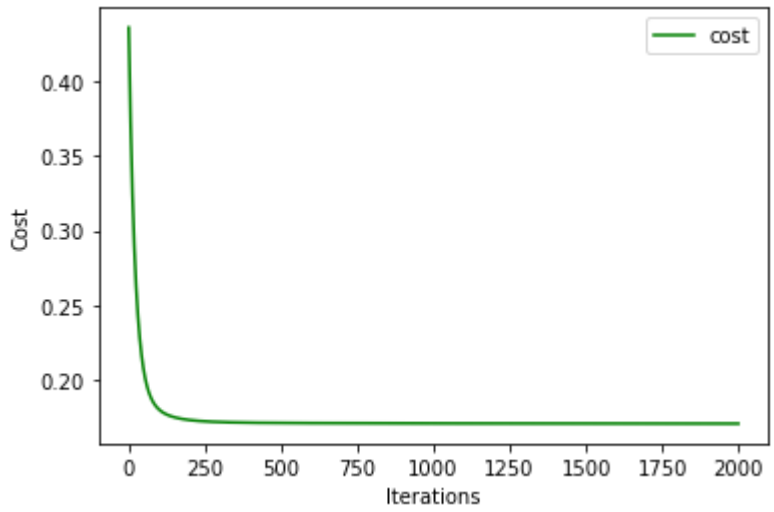

```
Regularisation parameter 5e-06
Validation Set RMSE=> 0.8599830377739207
Tranning Set RMSE => 0.5833127613815478
```




```
Regularisation parameter 0.0006
Validation Set RMSE=> 0.9027627515683939
Tranning Set RMSE => 0.5998633095266118
```



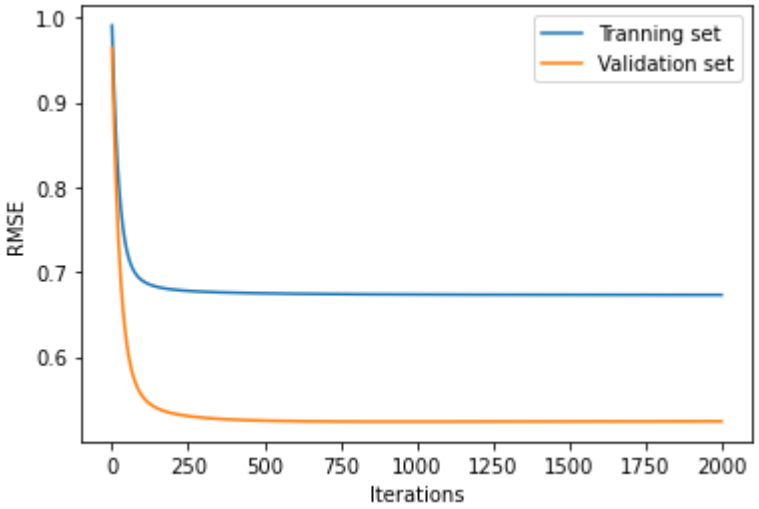

```
Regularisation parameter 7e-05
Validation Set RMSE=> 0.8673471706837934
Tranning Set RMSE => 0.5844403951382474
```

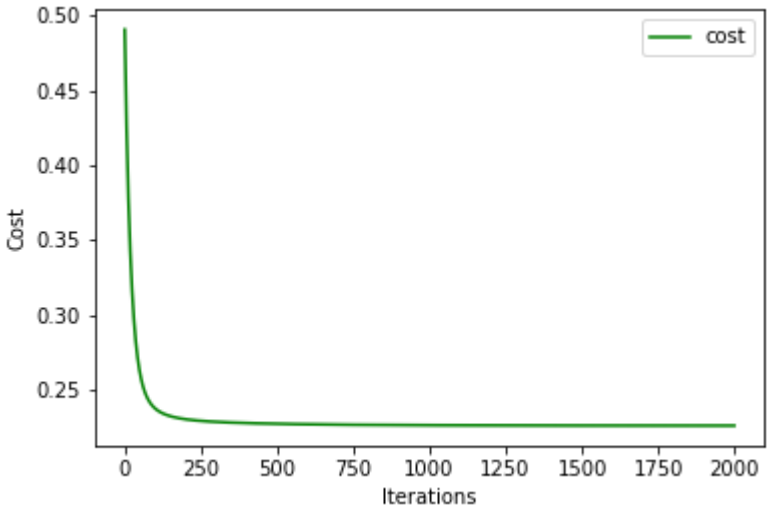
The best parameter is 0.01

In [16]:

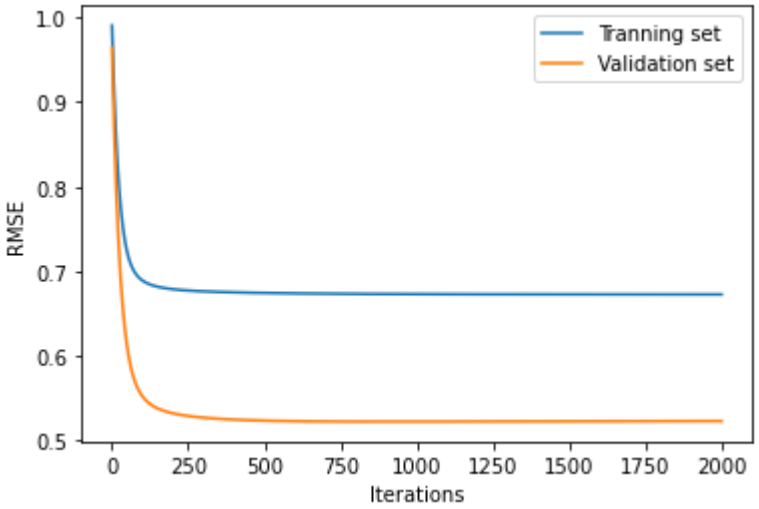
```
for i in range(5,6):
    l=[]
    l1=[]
    X_Trains, Y_Trains, X_Tests, Y_Tests = K_fold_validation(i,norm(pd.read_csv("Real e
    for j in range(i):
        mp=[0.01,0.001,0.000005,0.0006,0.00007]
        for m in range(5):
            model1=Linear_Regression_ridge(0.01,2000,mp[m])
            model1.fit(X_Trains[j],Y_Trains[j],X_Tests[j],Y_Tests[j])

            #we got the best split at k=5 and the 1th split so only showing that one
            if(j==1):

                print("Regularisation parameter {}".format(mp[m]))
                print("Validation Set RMSE=> {}".format(model1.val_err[-1]))
                print("Tranning Set RMSE => {}".format(model1.tr_err[-1]))
                model1.plot1()
```

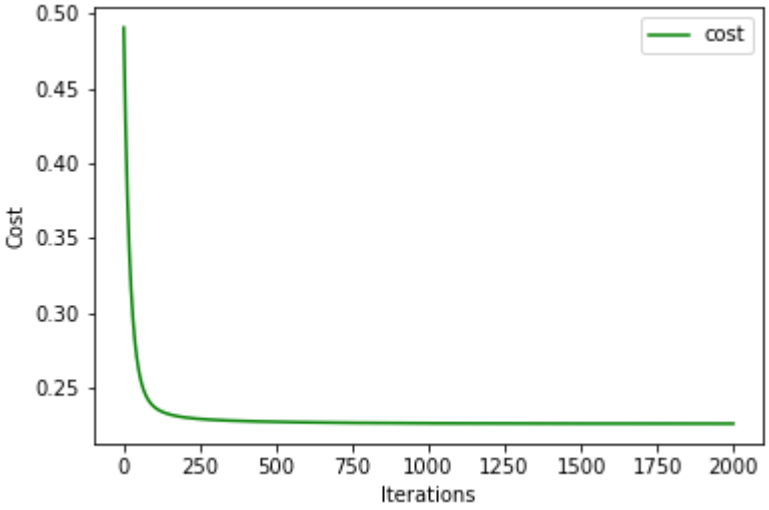
```
Regularisation parameter 0.01
Validation Set RMSE=> 0.5239991291862338
Tranning Set RMSE => 0.6730139364760966
```



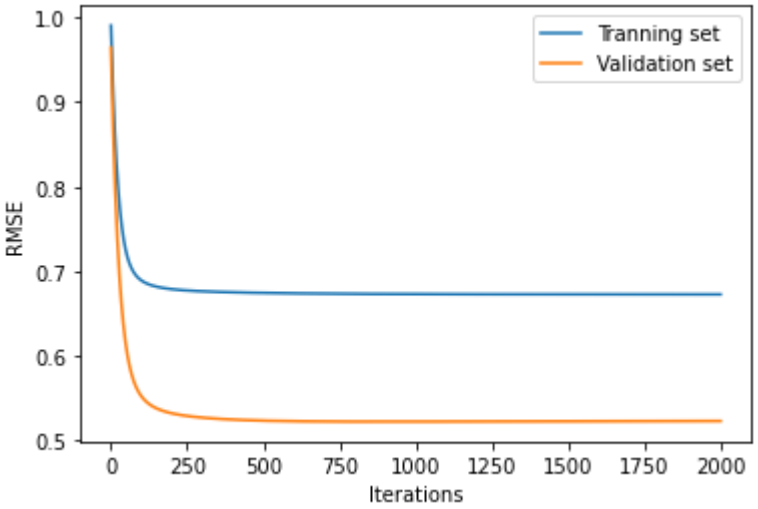



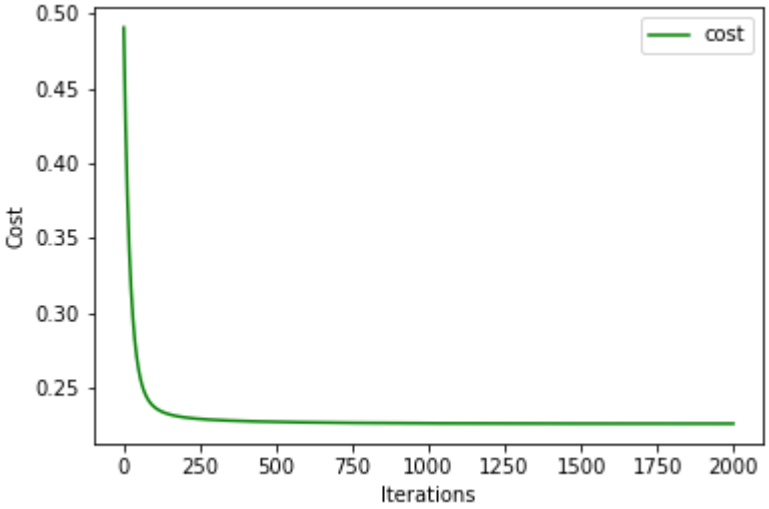

```
Regularisation parameter 0.001
Validation Set RMSE=> 0.5232712602950086
Tranning Set RMSE => 0.6728304370646259
```



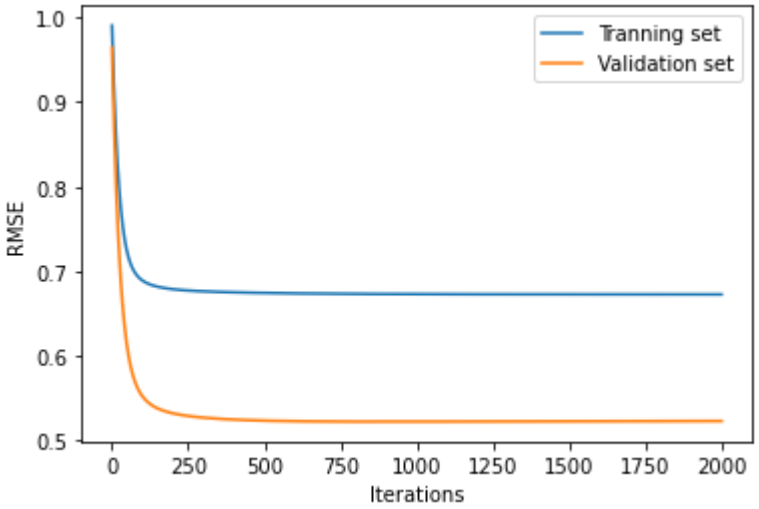


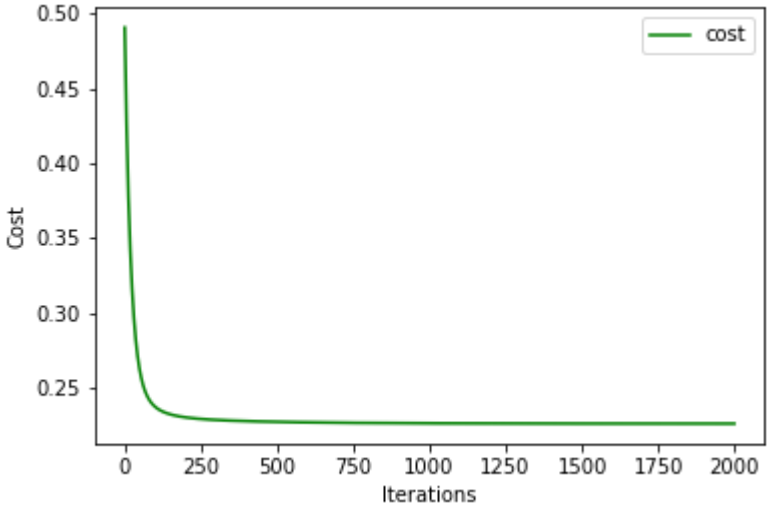

```
Regularisation parameter 5e-06
Validation Set RMSE=> 0.523205503196435
Tranning Set RMSE => 0.6728216809390176
```



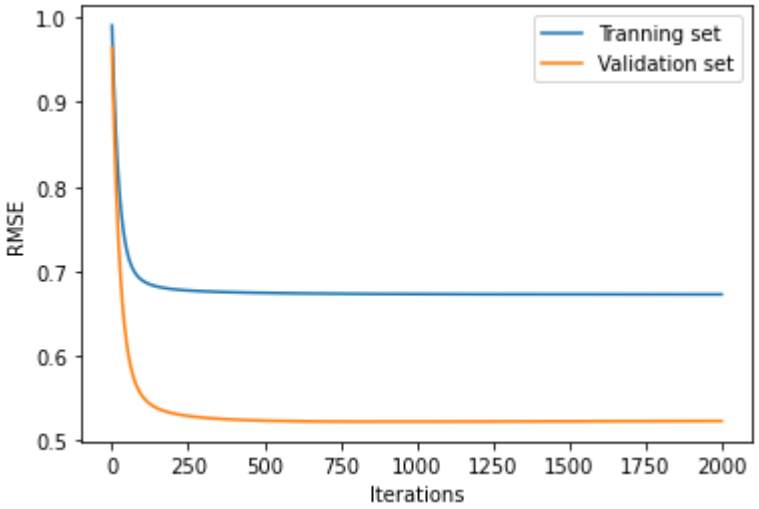



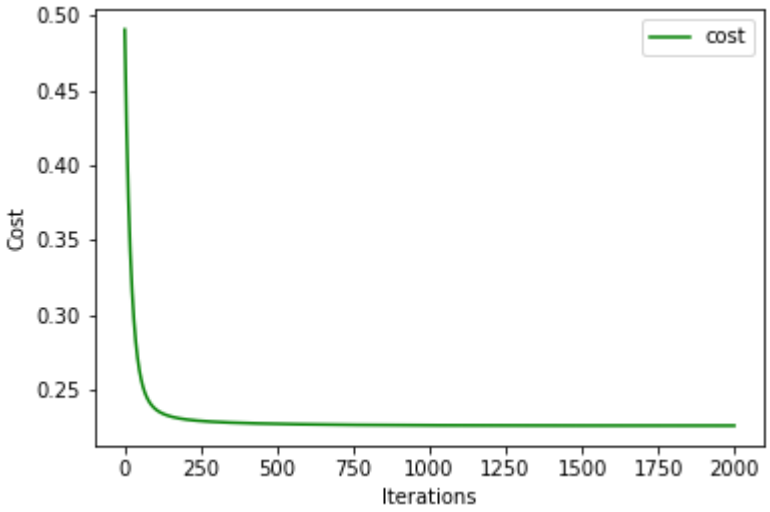

```
Regularisation parameter 0.0006
Validation Set RMSE=> 0.5232444313708448
Tranning Set RMSE => 0.672826604880376
```




```
Regularisation parameter 7e-05
Validation Set RMSE=> 0.5232096981710419
Tranning Set RMSE => 0.6728221730869017
```






The best parameter is 0.005

In [25]:

```
class LR_NORM():
    def __init__(self):
        self.name=0
    def fit(self, X, Y):

        self.m,self.n = X.shape
        on=np.ones((self.m,1))
        self.X=np.concatenate((on,X),axis=1)
        self.n=self.n+1
        self.Y=Y
        self.normal_eq()

    def normal_eq(self):
        self.W = np.linalg.inv(self.X.T.dot(self.X)).dot(self.X.T).dot(self.Y)

    def predict(self,X):
        X=np.array(X)
        return X.dot(self.W[1:])+self.W[0]
```

In [26]:

```
for i in range(2,6):
    X_Trains, Y_Trains, X_Tests, Y_Tests = K_fold_validation(i,norm(pd.read_csv("Real e
    l_c=[]
    for j in range(i):
        model=LR_NORM()
        model.fit(X_Trains[j],Y_Trains[j])
        y_pr=model.predict(X_Tests[j])
        l_c.append(np.sqrt(np.mean(np.square(model.predict(X_Tests[j])-Y_Tests[j]))))
        print("Rmse for k={} and split ={} =>".format(i,j)+str(sum(l_c)/len(l_c)))
    print()
```

```
Rmse for k=2 and split =0 =>0.6756446887548269
Rmse for k=2 and split =1 =>0.6604265204712334
```

```
Rmse for k=3 and split =0 =>0.6541133917854346
Rmse for k=3 and split =1 =>0.6976576051546526
Rmse for k=3 and split =2 =>0.6537020357875981

Rmse for k=4 and split =0 =>0.6770590790553567
Rmse for k=4 and split =1 =>0.6281281547675499
Rmse for k=4 and split =2 =>0.6191999641902847
Rmse for k=4 and split =3 =>0.6602538105727593

Rmse for k=5 and split =0 =>0.8688665619458582
Rmse for k=5 and split =1 =>0.7588676235103131
Rmse for k=5 and split =2 =>0.7204739580173932
Rmse for k=5 and split =3 =>0.663132149920377
Rmse for k=5 and split =4 =>0.6483658795405521
```

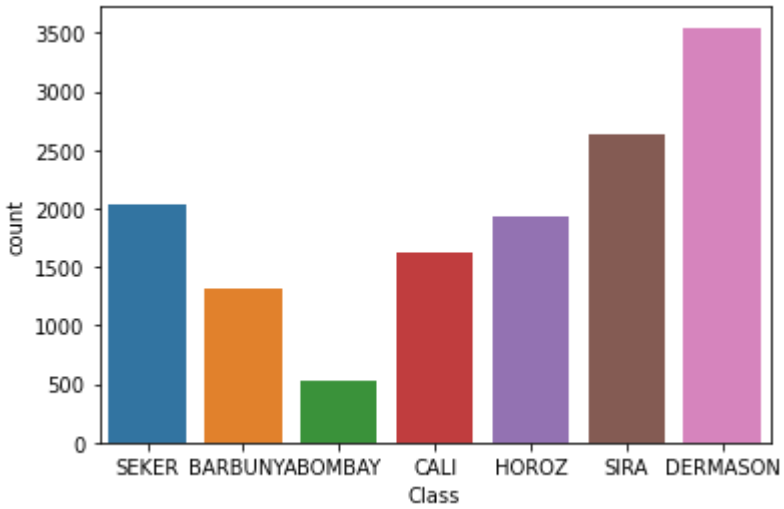
In [ ]:

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.manifold import TSNE
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB ,MultinomialNB
```

In [4]:
```
df = pd.read_excel('Dry_Bean_Dataset.xlsx')
```

# Part a

In [5]:
```
sns.countplot(x='Class', data=df, )
plt.show()
```




Analysis:

1) Major data entries are from DERMASON Class

2) Least number of data entries are from BOMBAY Class

3) A model Trained on this dataset will be more provide best predictions for DERMASON and, worst Predictions for BOMBAY Class According of LLN

## Part b

Perform EDA (histograms, box plots, scatterplots, etc.) and give at least five insights on the data. Check the missing values in the dataset.

In [6]:
```
#datafram shape and size
```

```
df.shape
```

Out[6]:
```
(13611, 17)
```

In [7]:
```
#Column names
df.columns
```

Out[7]:
```
Index(['Area', 'Perimeter', 'MajorAxisLength', 'MinorAxisLength',
       'AspectRation', 'Eccentricity', 'ConvexArea', 'EquivDiameter', 'Extent',
       'Solidity', 'roundness', 'Compactness', 'ShapeFactor1', 'ShapeFactor2',
       'ShapeFactor3', 'ShapeFactor4', 'Class'],
      dtype='object')
```

In [8]:
```
#data information
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13611 entries, 0 to 13610
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Area             13611 non-null  int64  
 1   Perimeter        13611 non-null  float64
 2   MajorAxisLength  13611 non-null  float64
 3   MinorAxisLength  13611 non-null  float64
 4   AspectRation     13611 non-null  float64
 5   Eccentricity     13611 non-null  float64
 6   ConvexArea       13611 non-null  int64  
 7   EquivDiameter    13611 non-null  float64
 8   Extent           13611 non-null  float64
 9   Solidity         13611 non-null  float64
 10  roundness        13611 non-null  float64
 11  Compactness      13611 non-null  float64
 12  ShapeFactor1     13611 non-null  float64
 13  ShapeFactor2     13611 non-null  float64
 14  ShapeFactor3     13611 non-null  float64
 15  ShapeFactor4     13611 non-null  float64
 16  Class            13611 non-null  object 
dtypes: float64(14), int64(2), object(1)
memory usage: 1.8+ MB
```

## Insight

No Null Values in dataset

Number of entries in Dataset = 13611

All the features are float except convex area

In [11]:
```
#Attirubute columns
# ['Area', 'Perimeter', 'MajorAxisLength', 'MinorAxisLength','AspectRation',
#  'Eccentricity', 'ConvexArea', 'EquivDiameter', 'Extent','Solidity',
#  'roundness', 'Compactness', 'ShapeFactor1', 'ShapeFactor2','ShapeFactor3',
#  'ShapeFactor4']
```

In [12]:
```
#target column
```

```
# ['Class']
```

In [13]:

```
#Top 5 entries in the dataframe
df.head()
```

Out[13]:

| | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | ConvexArea | Equiv |
|---|---|---|---|---|---|---|---|---|
| **0** | 28395 | 610.291 | 208.178117 | 173.888747 | 1.197191 | 0.549812 | 28715 | 19 |
| **1** | 28734 | 638.018 | 200.524796 | 182.734419 | 1.097356 | 0.411785 | 29172 | 19 |
| **2** | 29380 | 624.110 | 212.826130 | 175.931143 | 1.209713 | 0.562727 | 29690 | 19 |
| **3** | 30008 | 645.884 | 210.557999 | 182.516516 | 1.153638 | 0.498616 | 30724 | 19 |
| **4** | 30140 | 620.134 | 201.847882 | 190.279279 | 1.060798 | 0.333680 | 30417 | 19 |

In [14]:

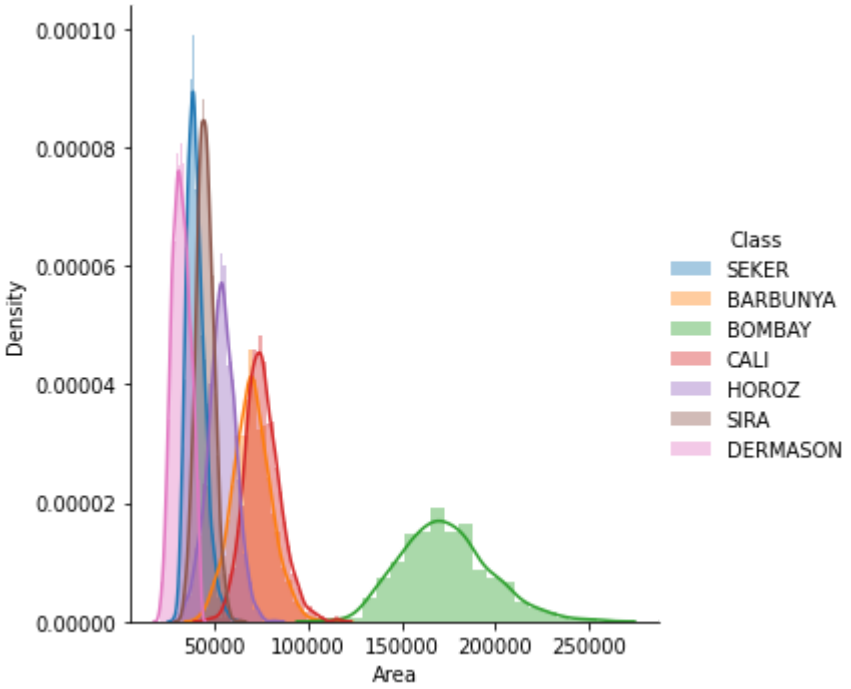
```
#dataframe description
df.describe()
```

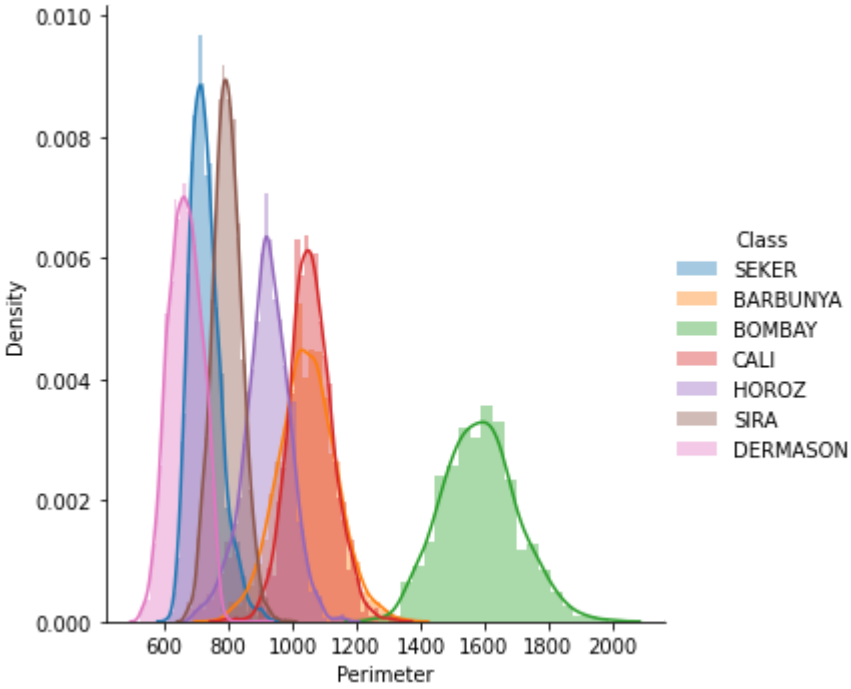
Out[14]:

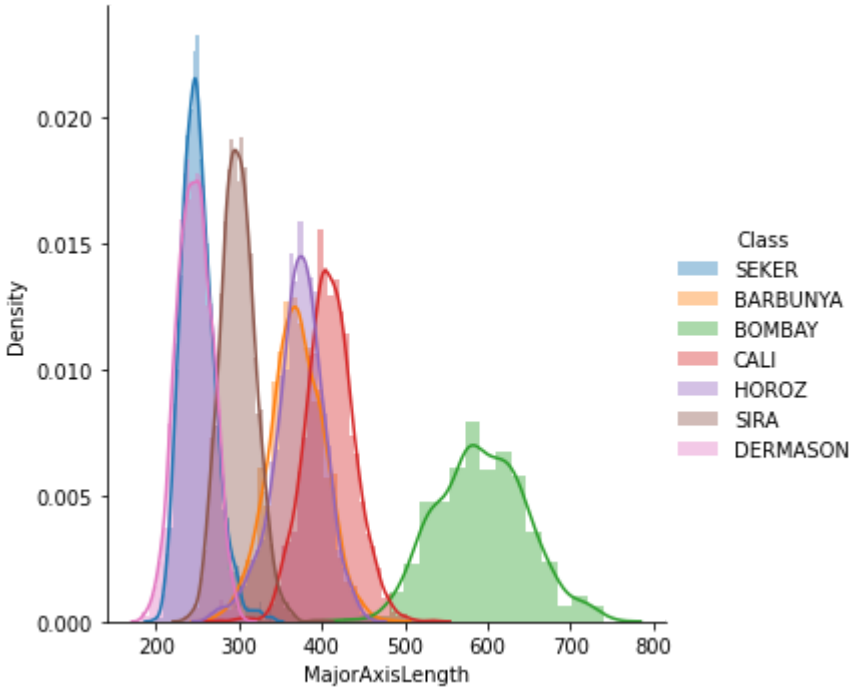
| | Area | Perimeter | MajorAxisLength | MinorAxisLength | AspectRation | Eccentricity | |
|---|---|---|---|---|---|---|---|
| **count** | 13611.000000 | 13611.000000 | 13611.000000 | 13611.000000 | 13611.000000 | 13611.000000 | 1 |
| **mean** | 53048.284549 | 855.283459 | 320.141867 | 202.270714 | 1.583242 | 0.750895 | 5 |
| **std** | 29324.095717 | 214.289696 | 85.694186 | 44.970091 | 0.246678 | 0.092002 | 2 |
| **min** | 20420.000000 | 524.736000 | 183.601165 | 122.512653 | 1.024868 | 0.218951 | 2 |
| **25%** | 36328.000000 | 703.523500 | 253.303633 | 175.848170 | 1.432307 | 0.715928 | 3 |
| **50%** | 44652.000000 | 794.941000 | 296.883367 | 192.431733 | 1.551124 | 0.764441 | 4 |
| **75%** | 61332.000000 | 977.213000 | 376.495012 | 217.031741 | 1.707109 | 0.810466 | 6 |
| **max** | 254616.000000 | 1985.370000 | 738.860153 | 460.198497 | 2.430306 | 0.911423 | 26 |

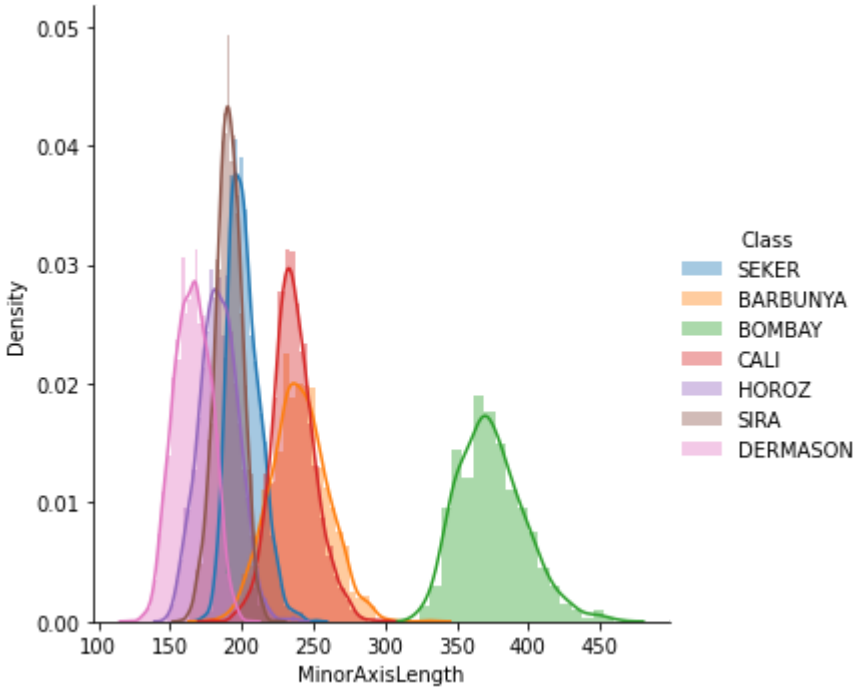
In [17]:

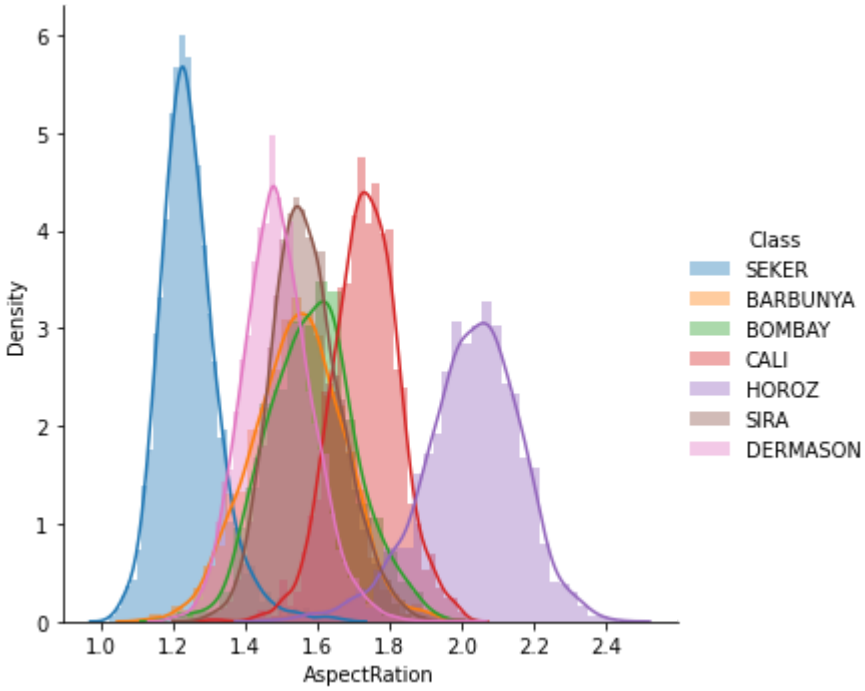
```
#Plotting the density curve for all the features of the dataset
for i in df.columns[:-1]:
    sns.FacetGrid(df, hue="Class", height=5).map(sns.distplot, i).add_legend()
#     sns.kdeplot(df[i],hue=df['Class'],fill=True,)
#     plt.show()
```

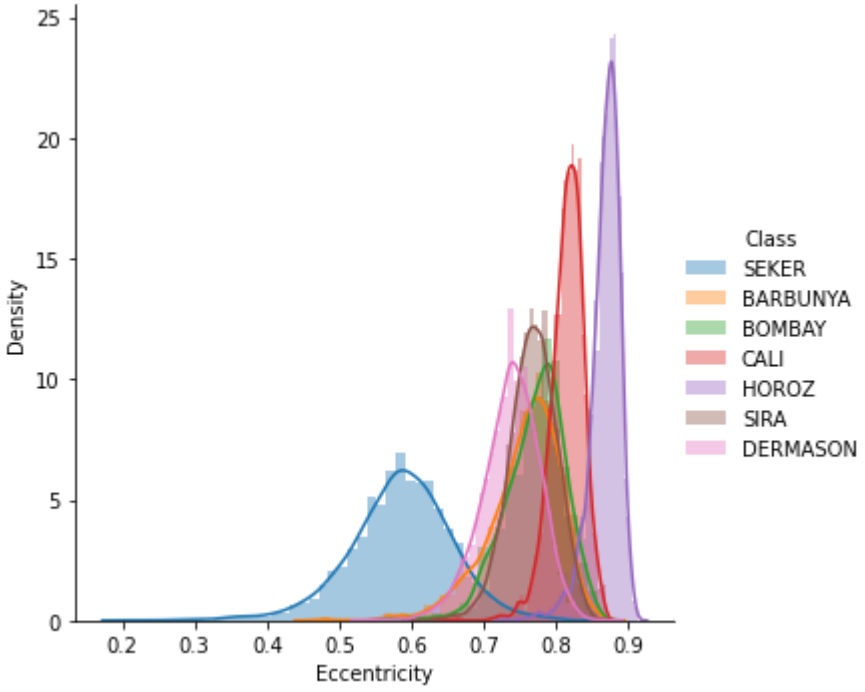

0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Density
50000
100000
150000
200000
250000
Area
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

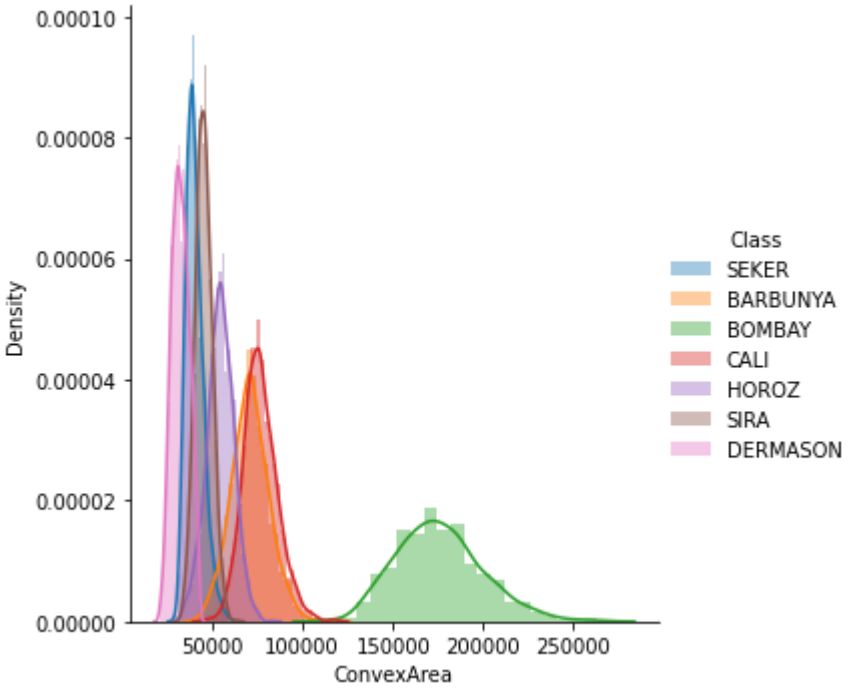

0.010
0.008
0.006
0.004
0.002
0.000
Density
600
800
1000
1200
1400
1600
1800
2000
Perimeter
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

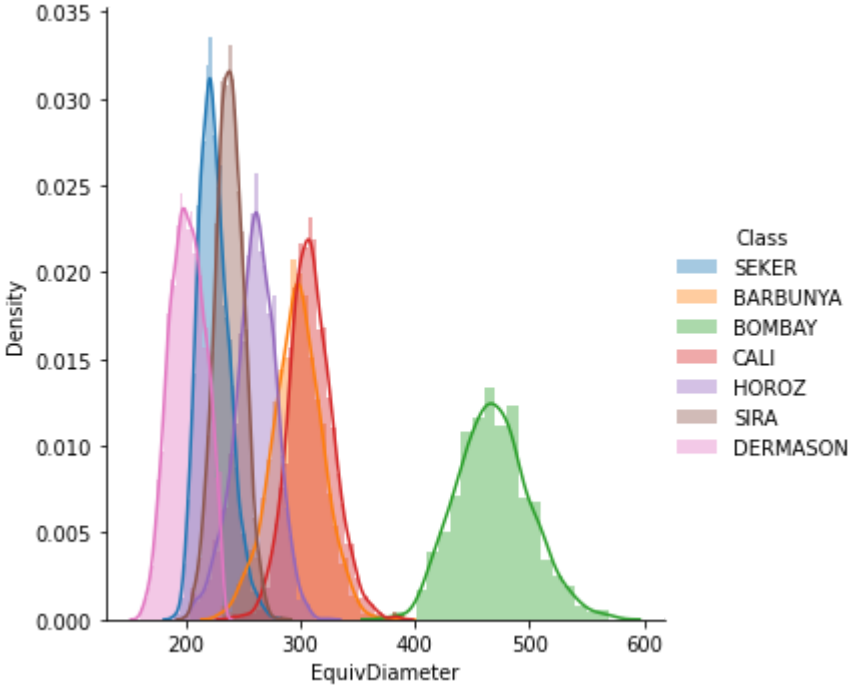

0.020
0.015
0.010
0.005
0.000
Density
200
300
400
500
600
700
800
MajorAxisLength
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

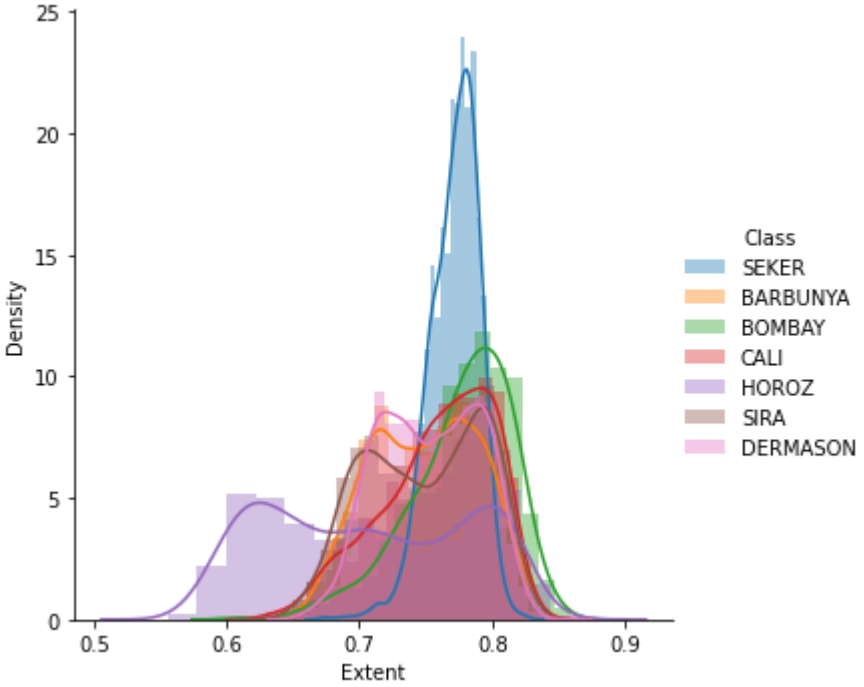

0.05
0.04
0.03
0.02
0.01
0.00
Density
100
150
200
250
300
350
400
450
MinorAxisLength
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

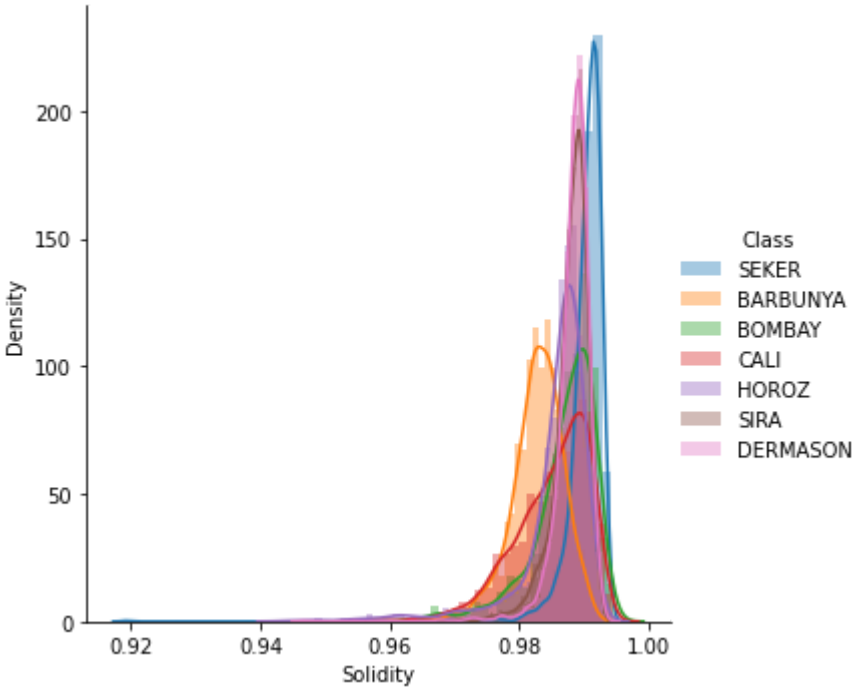

Density
AspectRation
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

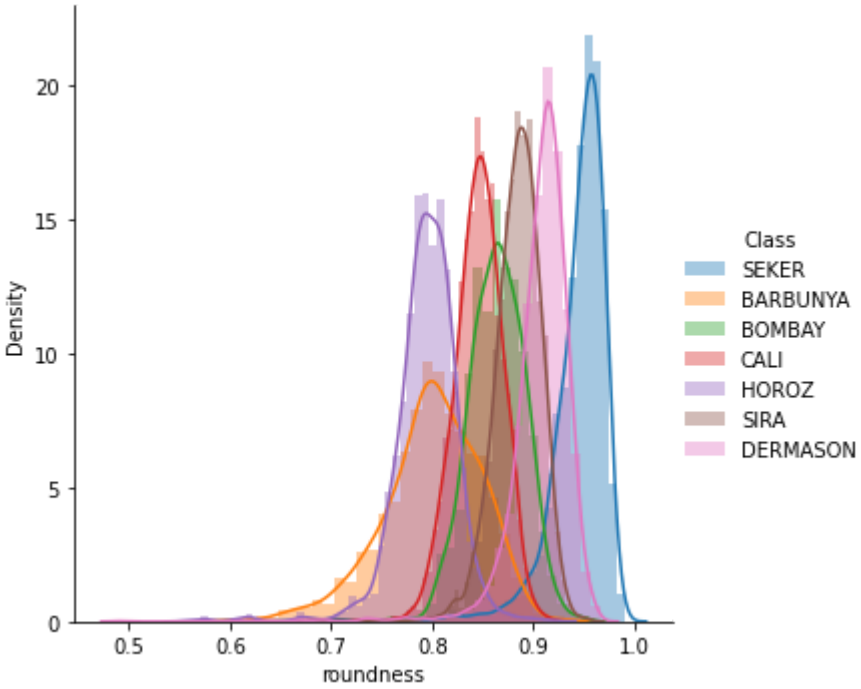

Density
Eccentricity
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

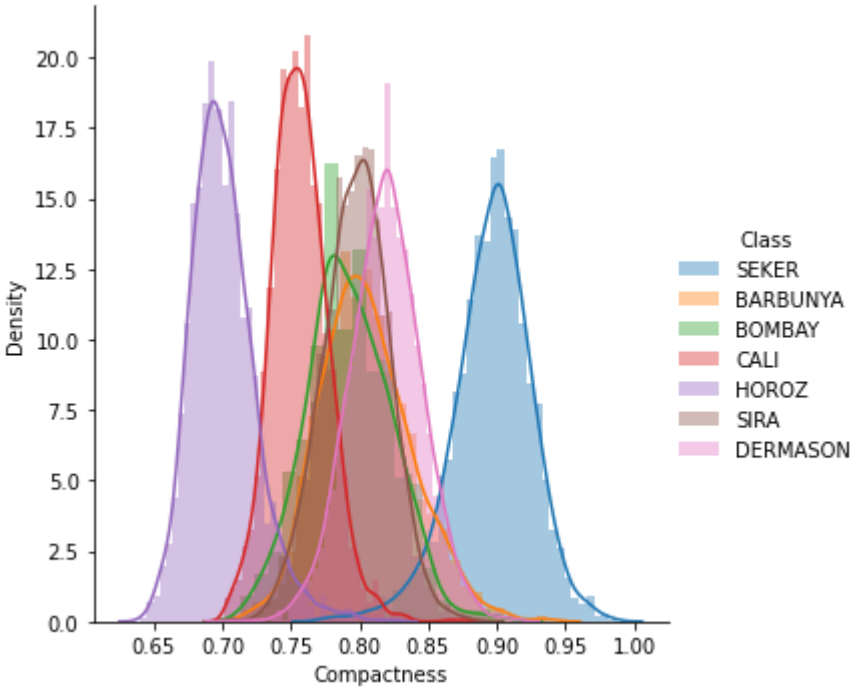

0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Density
50000
100000
150000
200000
250000
ConvexArea
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

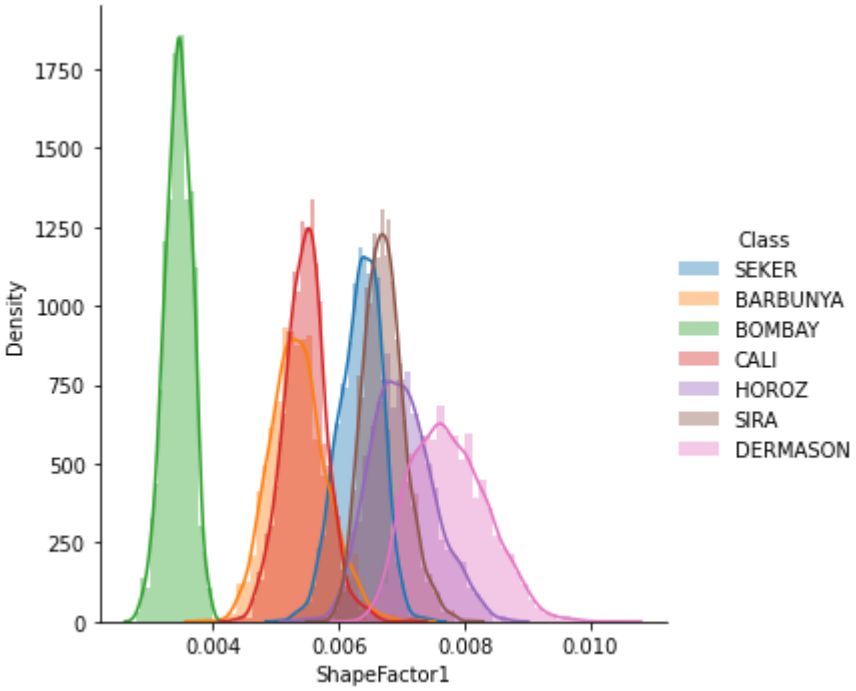

0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Density
200
300
400
500
600
EquivDiameter
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

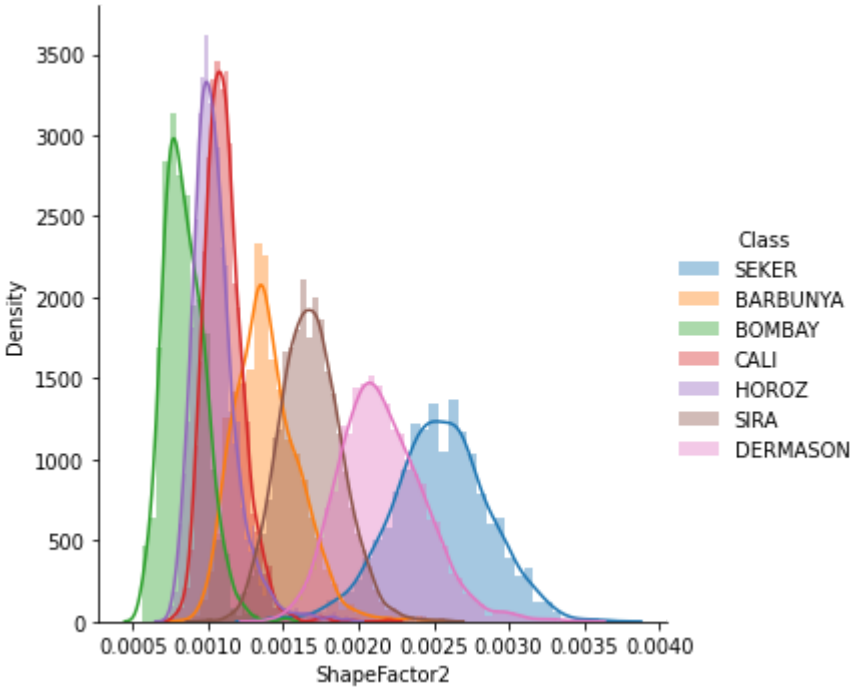

Density
Extent
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

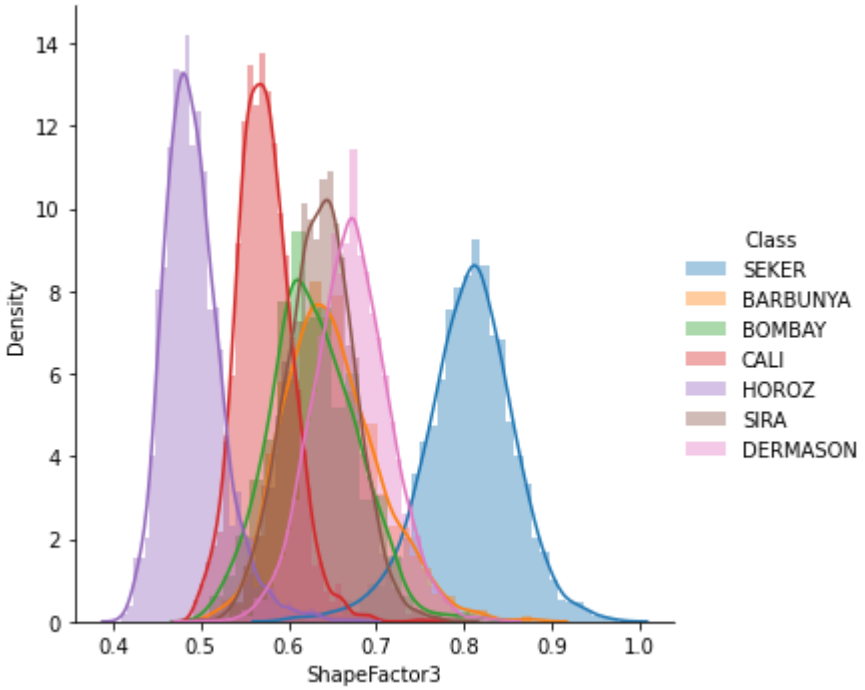

Density
Solidity
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

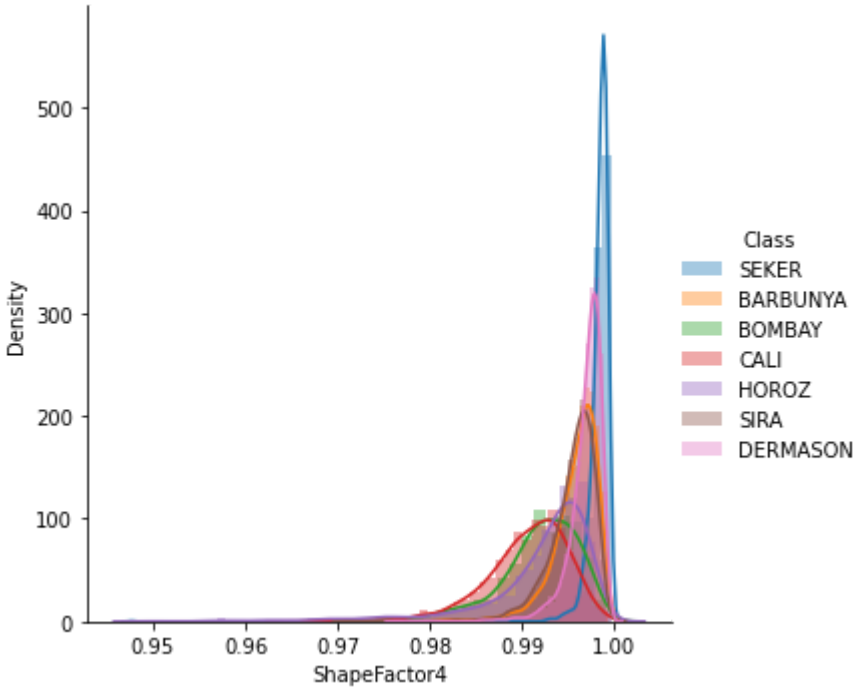

Density
roundness
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


Density
Compactness
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


Density
ShapeFactor1
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON


Density
ShapeFactor2
Class
SEKER
BARBUNYA
BOMBAY
CALI
HOROZ
SIRA
DERMASON

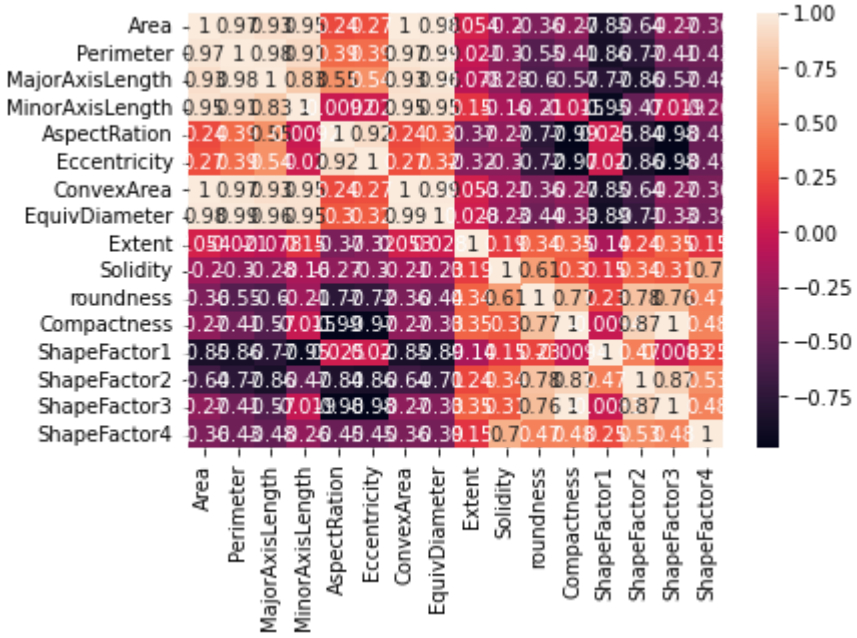



## INSIGHTS

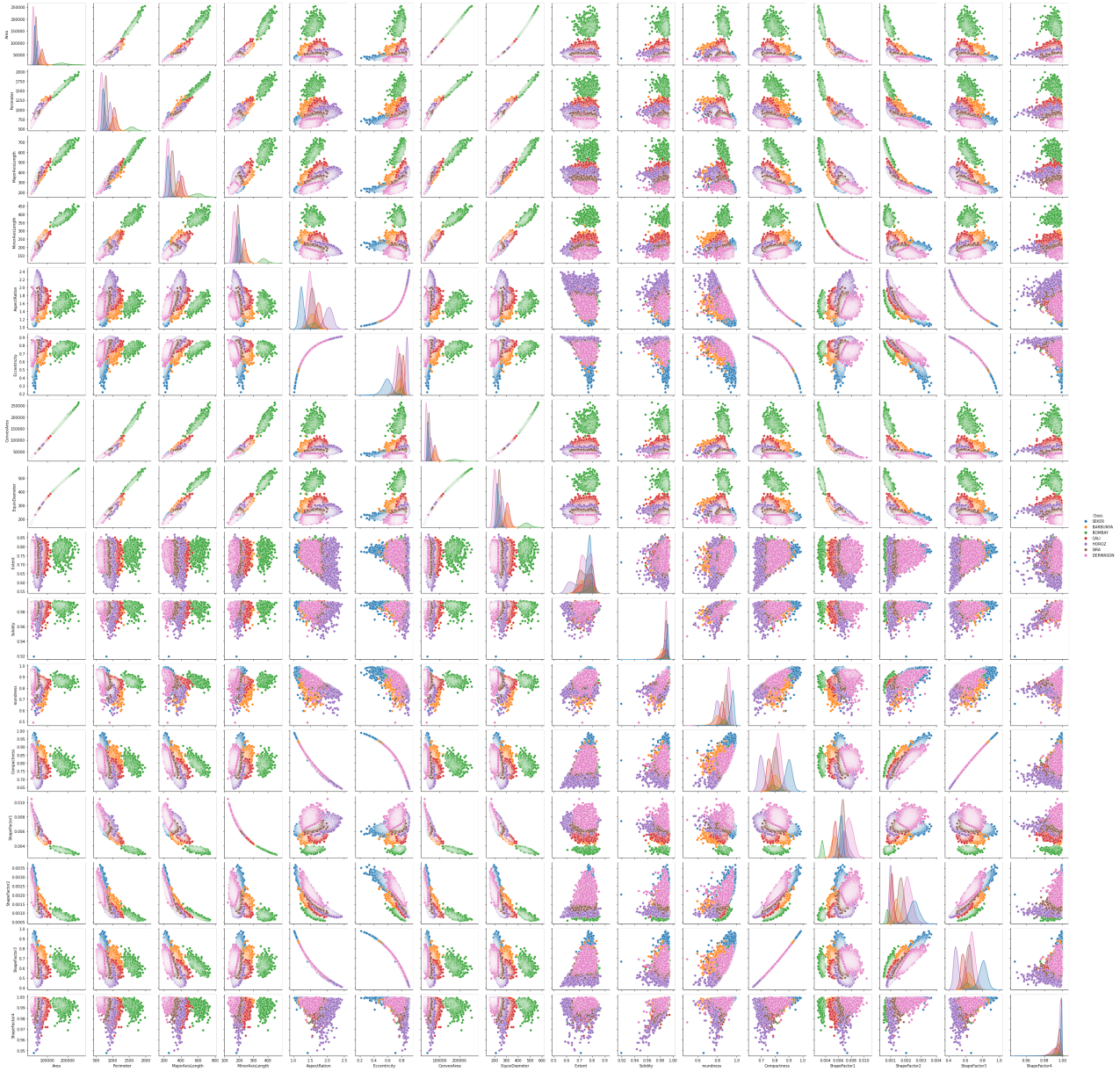
1. Beans from Bombay have higher area perimeter and major,minor axis length.
2. The shape factor 2,3 4 are highest in 'SEKER' class.
3. All the classes have comparable extent feature.

In [18]:
```
#correlation  heatmap to check for correlation
sns.heatmap(df.corr(method='pearson'),annot = True);

plt.show()
```




In [9]:
```
sns.pairplot(data=df,hue='Class')
```

Out[9]:
```
<seaborn.axisgrid.PairGrid at 0x1db9239d460>
```

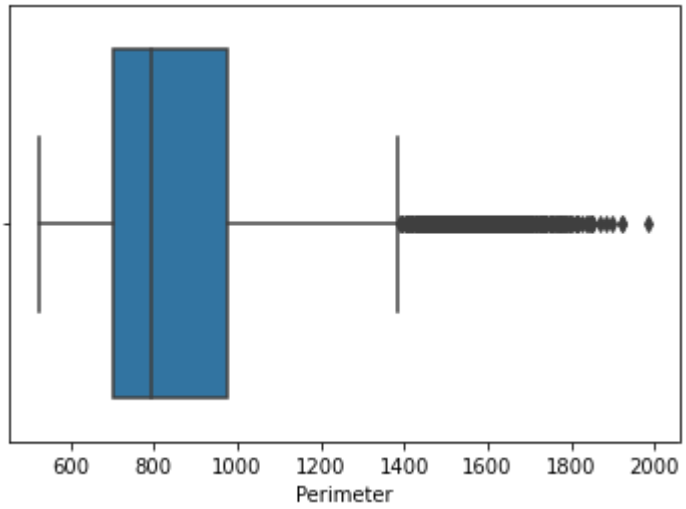


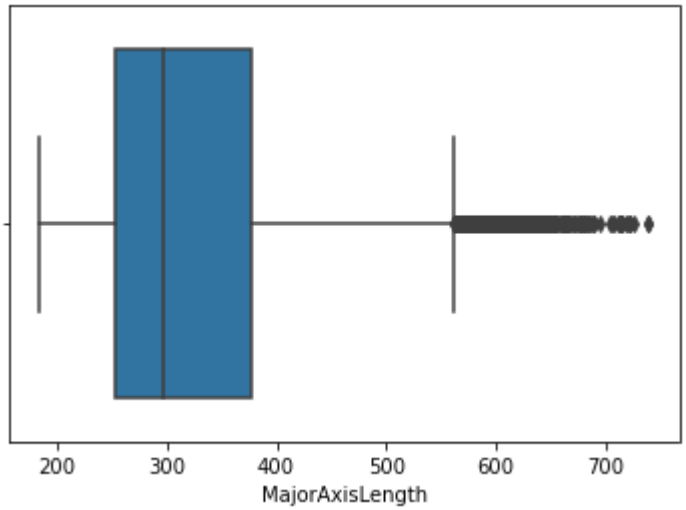
## Insights

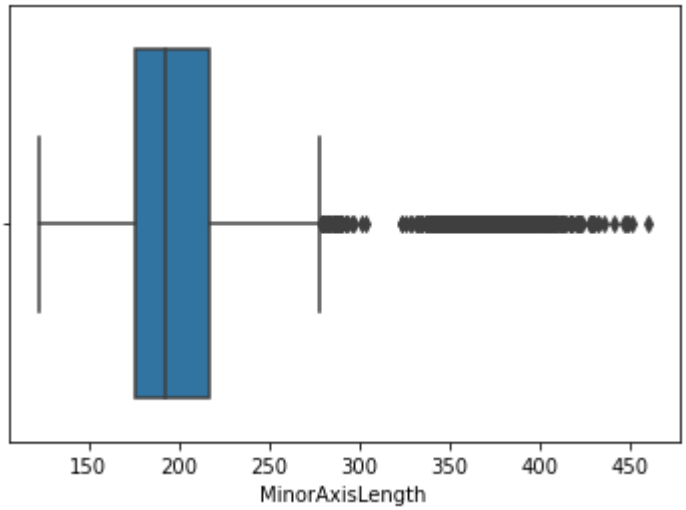
1. Area feature is positively highly correlated with convexarea and equidiameter, and negatively correlated with shapefactor 2 nd 3.
2. ShapeFactor 3 and compactness are positively correlated and other correlations are also there but not to the extremes

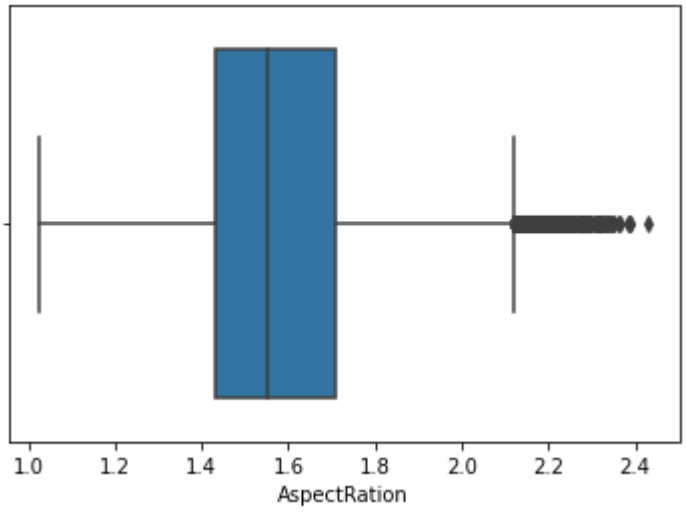
In [10]:

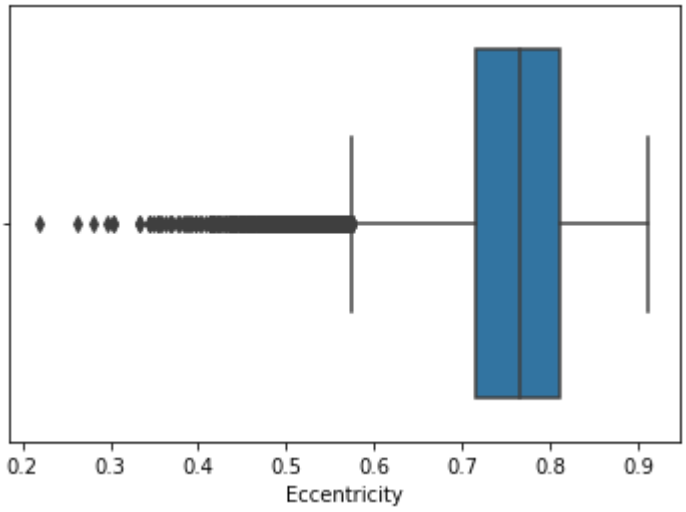
```
#box plots fot outlier finding
for i in df.columns[1:-1]:
    sns.boxplot(x=i, data=df)# outlier.add_legend()
    plt.show()
```

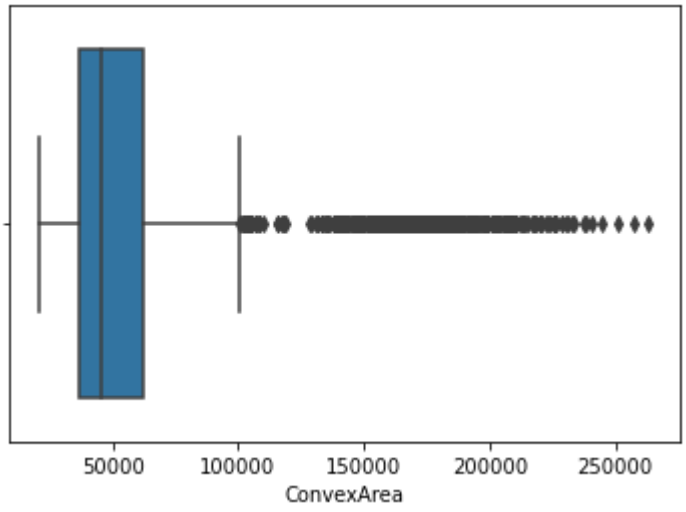

600
800
1000
1200
1400
1600
1800
2000
Perimeter

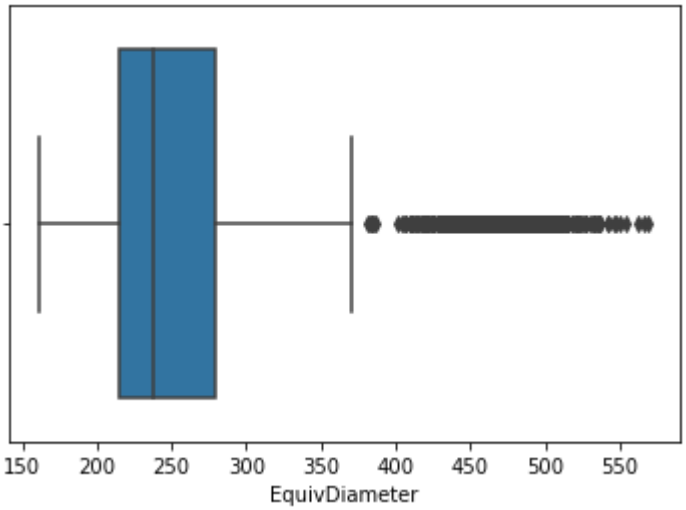

200
300
400
500
600
700
MajorAxisLength

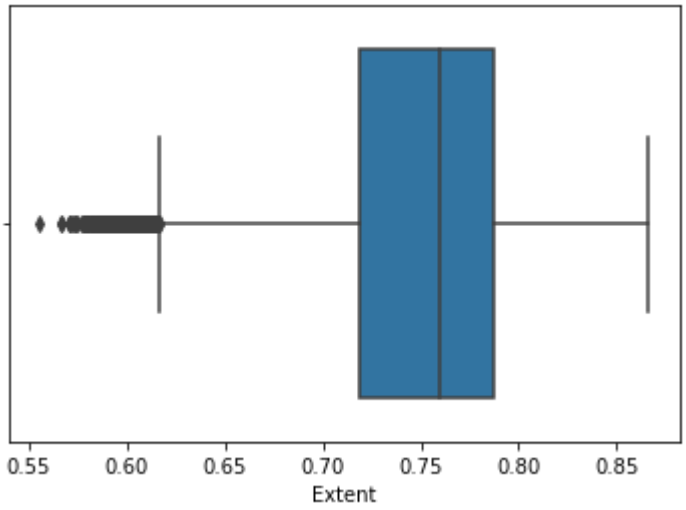

150
200
250
300
350
400
450
MinorAxisLength

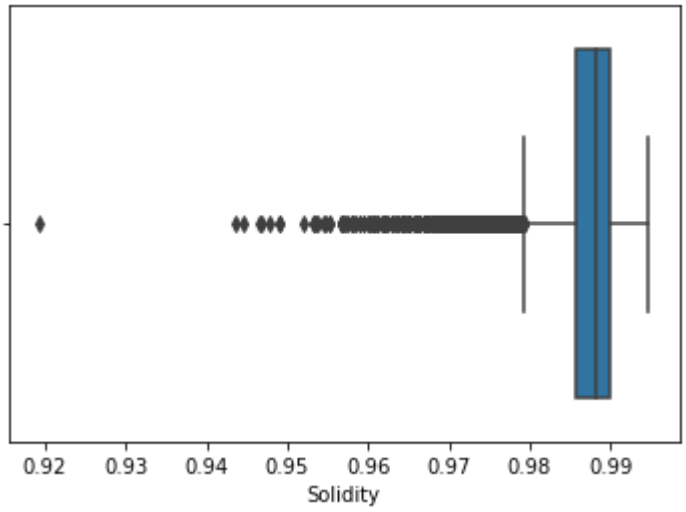

1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
AspectRation

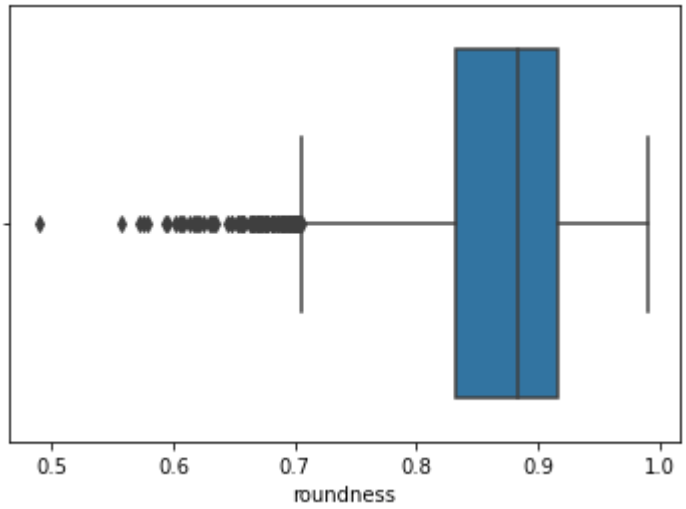

0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Eccentricity

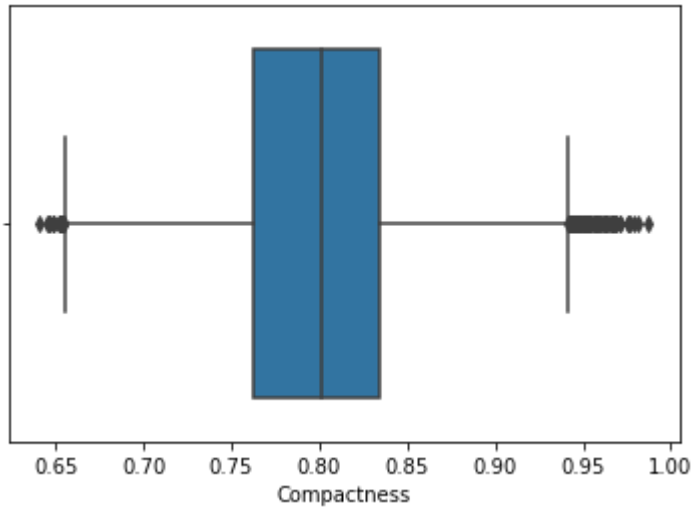

50000
100000
150000
200000
250000
ConvexArea

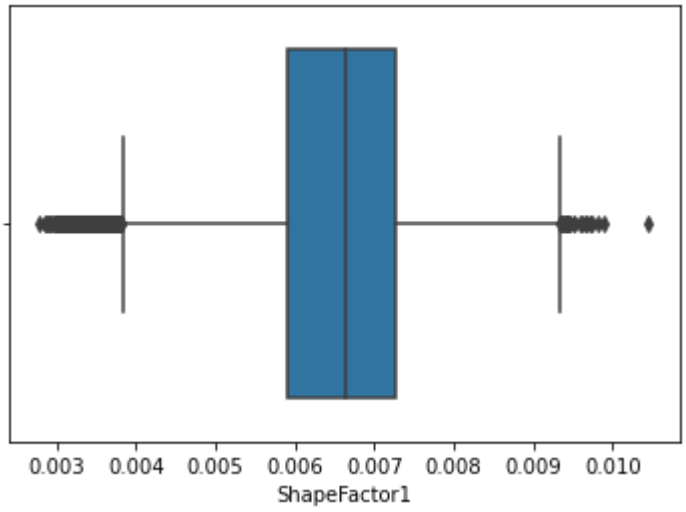

150
200
250
300
350
400
450
500
550
EquivDiameter

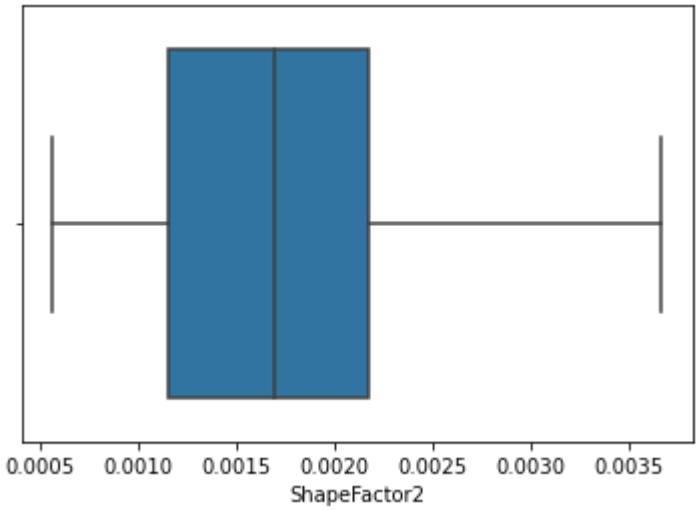

0.55
0.60
0.65
0.70
0.75
0.80
0.85
Extent

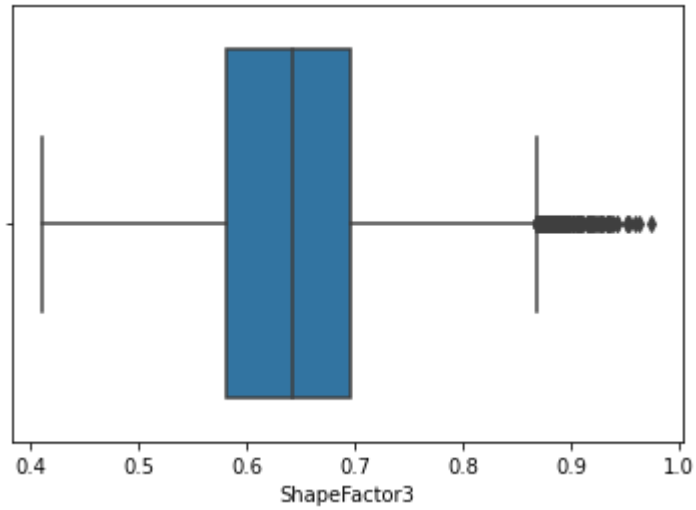

0.92
0.93
0.94
0.95
0.96
0.97
0.98
0.99
Solidity

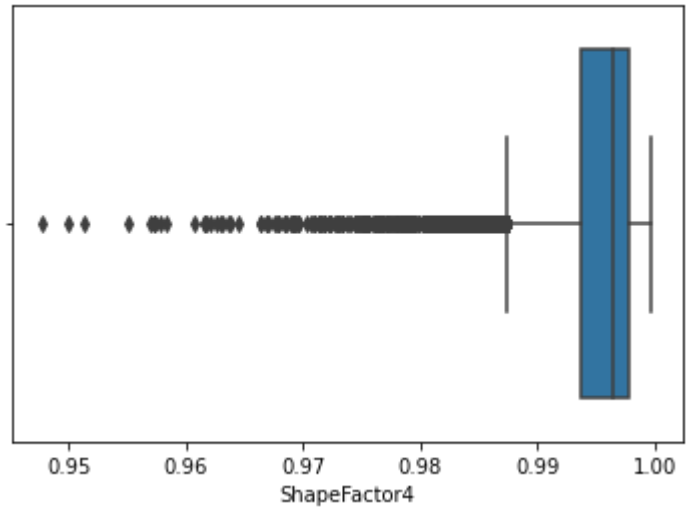

0.5
0.6
0.7
0.8
0.9
1.0
roundness


0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00
Compactness


0.003
0.004
0.005
0.006
0.007
0.008
0.009
0.010
ShapeFactor1




## Insights

As we can see from the box plots there are a number of outliear in the distribution
but these outliers can not be removed because removing them will delete a complete class from the distribution

## Part c

Use TSNE (t-distributed stochastic neighbor embedding) algorithm to reduce data dimensions to 2 and plot the resulting data as a scatter plot. Comment on the separability of the data

In [11]:
```
##Performing standardization on data before TSNE

#target variable
tar_var=df["Class"]
#Attributes values
attr_var=df.drop("Class",axis=1)

# standardization of attribute variables
standarized_data = StandardScaler().fit_transform(attr_var)
print(standarized_data.shape)
```

```
(13611, 16)
```

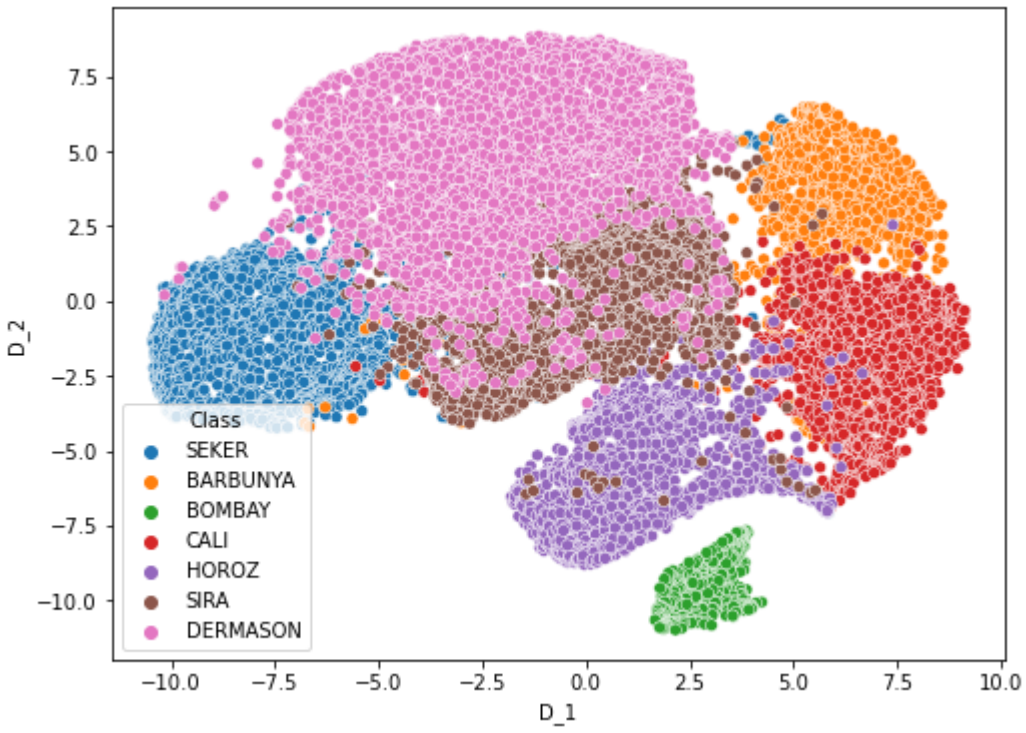
In [12]:
```
#performing TSNE with 300 iterations
tsne_model = TSNE(n_components=2, random_state=0,perplexity=60, n_iter=300)
tsne_data = tsne_model.fit_transform(standarized_data)
print(tsne_data.shape)
```

```
(13611, 2)
```

In [13]:
```
#adding the target column to visualise the plots
tsne_1=np.vstack((tsne_data.T,tar_var)).T
tsne_1=pd.DataFrame(data=tsne_1,columns=("D_1","D_2","Class"))
```

In [14]:
```
plt.figure(figsize=(8,6))
sns.scatterplot(x="D_1",y="D_2",hue="Class",data=tsne_1)
```

Out[14]:
```
<AxesSubplot:xlabel='D_1', ylabel='D_2'>
```



### Comment on the Data Separablity

Data is Separable it can be seen in the above plot

## Part D

Run the sklearn's implementation of Naive Bayes (Any 2 of your choice - refer here ). Report Accuracy, Recall, and Precision. Comment on the results and their differences from the two implementations of Naive Bayes. (80:20 train test split)

In [15]:
```
#encoding the class names to convert them to intergers

#creating labelEncoder
le = preprocessing.LabelEncoder()
# Converting string labels into numbers.
wheather_encoded=le.fit_transform(df["Class"])
df1=df
df1["Class"]=wheather_encoded
```

In [16]:
```
#target variable
tar_var=df1["Class"]
#Attributes values
attr_var=df1.drop("Class",axis=1)
```

In [17]:
```
#splitting the data into tranning and testing data with the
#testing data size to be 20%

X_train, X_test, y_train, y_test= train_test_split(df1.drop("Class",axis=1),
                                                   df1["Class"], test_size=0.2,
                                                   random_state=18)
```

In [18]:
```
#Bayes
#Create a Gaussian Classifier
gnb = GaussianNB()

#Train the model using the training sets
gnb.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = gnb.predict(X_test)
```

In [19]:
```
# Model Accuracy
print("Gaussian Naive Bayes")
print("Accuracy: ",metrics.accuracy_score(y_test, y_pred))
print("Precision Score: ",metrics.precision_score(y_test, y_pred,average='micro'))
print("Recall Score: ",metrics.recall_score(y_test, y_pred,average='micro'))
```

```
Gaussian Naive Bayes
Accuracy:  0.7627616599338964
Precision Score:  0.7627616599338964
Recall Score:  0.7627616599338964
```

In [20]:
```
#Bayes
#Create a Multinomial Classifier
mnb=MultinomialNB()

#Train the model using the training sets
mnb.fit(X_train, y_train)

#Predict the response for test dataset
y_predclf = mnb.predict(X_test)
```

In [21]:
```
print("Multinomial Naive Bayes")
print("Accuracy: ",metrics.accuracy_score(y_test, y_predclf))
print("Precision Score: ",metrics.precision_score(y_test, y_predclf,average='micro'))
print("Recall Score: ",metrics.recall_score(y_test, y_predclf,average='micro'))
```

```
Multinomial Naive Bayes
Accuracy:  0.7847961806830701
Precision Score:  0.7847961806830701
Recall Score:  0.7847961806830701
```

## Comment on the results

Since Multinomial Classfier is best suited for discrete value prediction and each features here can have a distribution type of it's own while Gaussian is better for predicting continuous values we got better results by multinomial bayes

## Part E

Use Principal Component Analysis (PCA) to reduce the number of features and use the reduced data set for model training. Retain different amounts of variance values, ranging from 0.9 to 1 in steps of 0.01. Compare results (Accuracy, Precision, Recall, and F-1 score). (80:20 train test split)

In [22]:
```
##PCA function using Naive Bayes
def PCA_f(dfx):
    x=dfx.drop('Class',axis=1)
    y=dfx['Class']

    #
    for i in range(4,13,2):

        #Using i/100 to decide the variance to be Kept
        #print(i/100)
        pca = PCA(n_components=i)
        pca.fit(x)
        data_pca = pca.transform(x)

        #naming the Columns PC+{i}
        column=[]
        for j in range(1,len(data_pca[0])+1):
            column.append("PC"+str(j))
        data_pca = pd.DataFrame(data_pca, columns=column)
```

```
        #dividing PCA data into train and test
        X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(
            data_pca,df["Class"], test_size=0.2,random_state=18)

        #Gaussian Classifier
        gnb_pca=GaussianNB()
        gnb_pca.fit(X_train_pca, y_train_pca)

        #predicting the Values
        y_pred=gnb_pca.predict(X_test_pca)

        #printing the results
        print()
        print("Gaussian Naive Bayes with PCA with {} Components".format(i))
        print("Accuracy: ",metrics.accuracy_score(y_test_pca, y_pred))
        print("Precision Score: ",metrics.precision_score(y_test_pca, y_pred,average='m
        print("Recall Score: ",metrics.recall_score(y_test_pca, y_pred,average='micro')
        print("F1 Score: ",metrics.f1_score(y_test_pca, y_pred,average='micro'))
```

In [23]:

```
# df2=df1
PCA_f(df1)
# print(df1)
```

```
Gaussian Naive Bayes with PCA with 4 Components
Accuracy:  0.8901946382666177
Precision Score:  0.8901946382666177
Recall Score:  0.8901946382666177
F1 Score:  0.8901946382666177

Gaussian Naive Bayes with PCA with 6 Components
Accuracy:  0.892398090341535
Precision Score:  0.892398090341535
Recall Score:  0.892398090341535
F1 Score:  0.892398090341535

Gaussian Naive Bayes with PCA with 8 Components
Accuracy:  0.892398090341535
Precision Score:  0.892398090341535
Recall Score:  0.892398090341535
F1 Score:  0.892398090341535

Gaussian Naive Bayes with PCA with 10 Components
Accuracy:  0.892398090341535
Precision Score:  0.892398090341535
Recall Score:  0.892398090341535
F1 Score:  0.892398090341535

Gaussian Naive Bayes with PCA with 12 Components
Accuracy:  0.892398090341535
Precision Score:  0.892398090341535
Recall Score:  0.892398090341535
F1 Score:  0.892398090341535
```

In [24]:

```
# import sys
# sys.setrecursionlimit(150)
```

## Part F

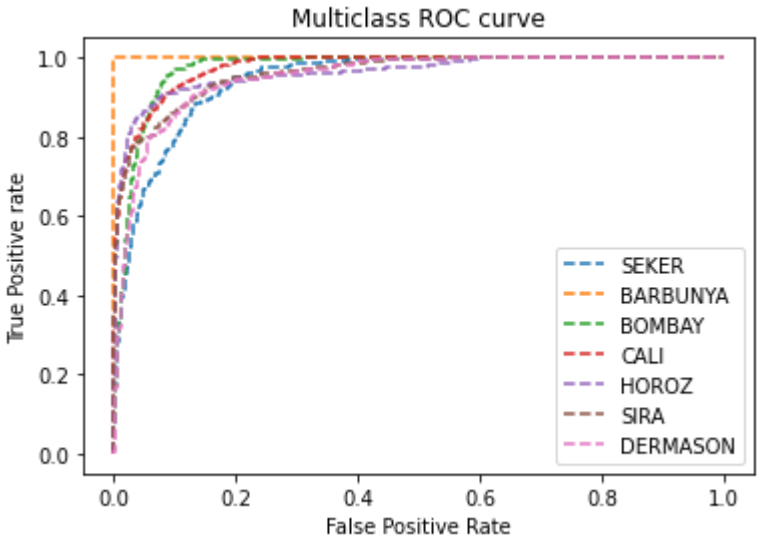
Use Scikit-learn to plot the ROC-AUC curves and comment on the out- put.

In [25]:
```
mnb1=MultinomialNB()
#splitting the data into tranning and testing data with the
#testing data size to be 20%

X_train, X_test, y_train, y_test= train_test_split(df.drop("Class",axis=1),
                                                   df["Class"], test_size=0.2,
                                                   random_state=18)


#tranning the model on tranning data
mnb1.fit(X_train,y_train)
```

Out[25]:
```
MultinomialNB()
```

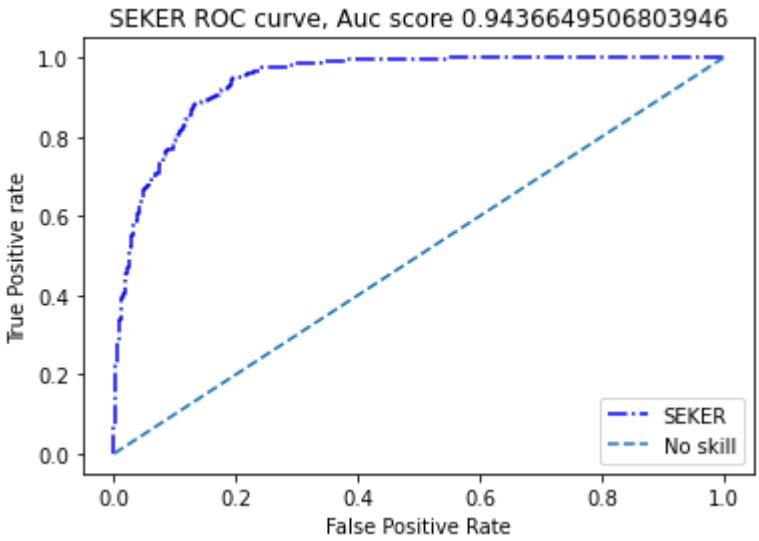
In [26]:
```
pred_prob = mnb1.predict_proba(X_test)

# roc curve for classes
fpr = {}
tpr = {}
thresh ={}
auc={}

n_class = 7

for i in range(n_class):
    fpr[i], tpr[i], _ = metrics.roc_curve(y_test, pred_prob[:,i], pos_label=i)




# plotting

cl_names=['SEKER', 'BARBUNYA', 'BOMBAY', 'CALI', 'HOROZ', 'SIRA', 'DERMASON']
for i in range(7):
  plt.plot(fpr[i], tpr[i], linestyle='--',label=cl_names[i])
plt.title('Multiclass ROC curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive rate')
plt.legend(loc='best')
```

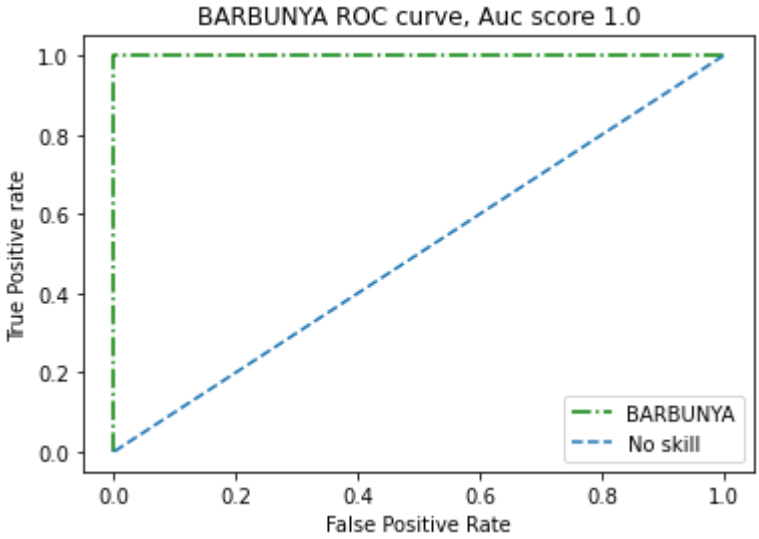
Out[26]:
```
<matplotlib.legend.Legend at 0x1dbb2c3df40>
```

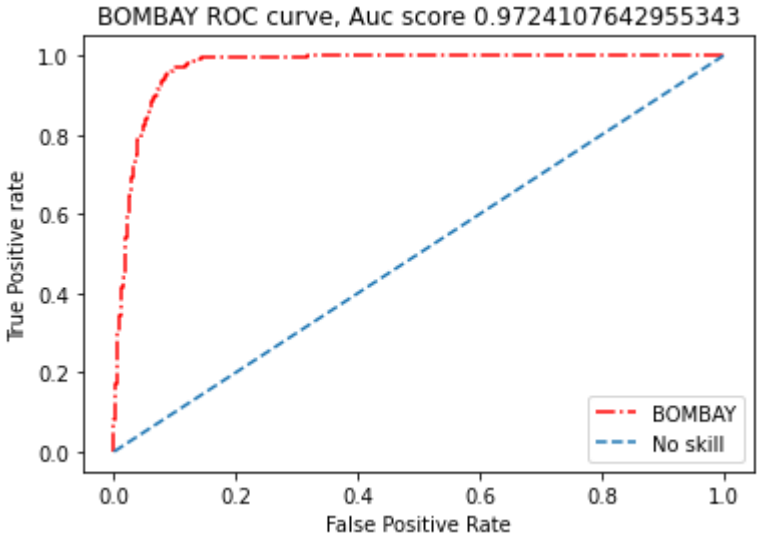


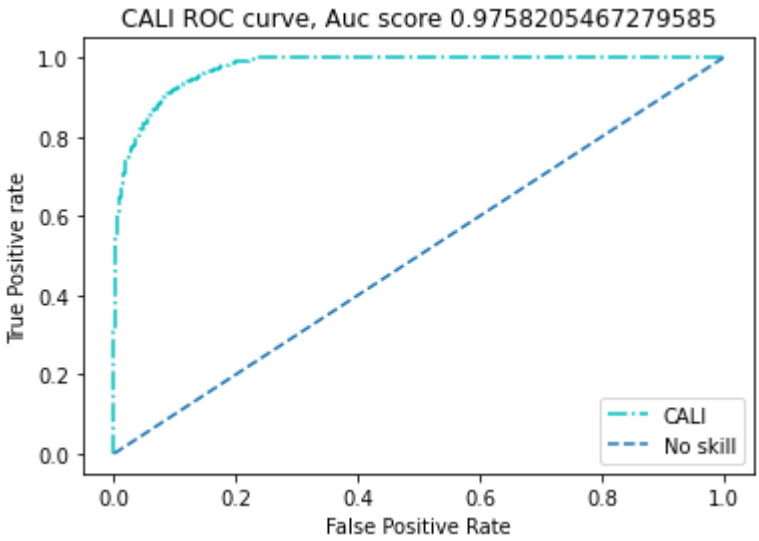

In [27]:

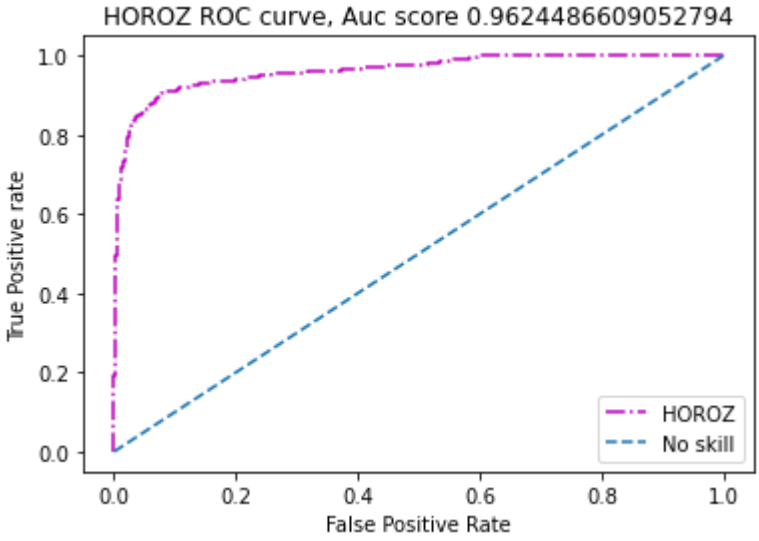
```
#Calculating the AUC score for every class
auc={}
for i in range(7):
  oc=[]
  for j in y_test:
    if i==j:
      oc.append(1)
    else:
      oc.append(0)
  auc[i]= metrics.roc_auc_score(oc, pred_prob[:,i])


colors=['b','g','r','c','m','k','y']
for i in range(7):
  plt.plot(fpr[i], tpr[i],color=colors[i], linestyle='-.',label=cl_names[i])

  plt.plot([0,1],[0,1],linestyle = "dashed" , label="No skill")

  plt.title(cl_names[i]+' ROC curve'+", Auc score "+str(auc[i]))
  plt.xlabel('False Positive Rate')
  plt.ylabel('True Positive rate')
  plt.legend(loc='lower right')
  plt.show()
```

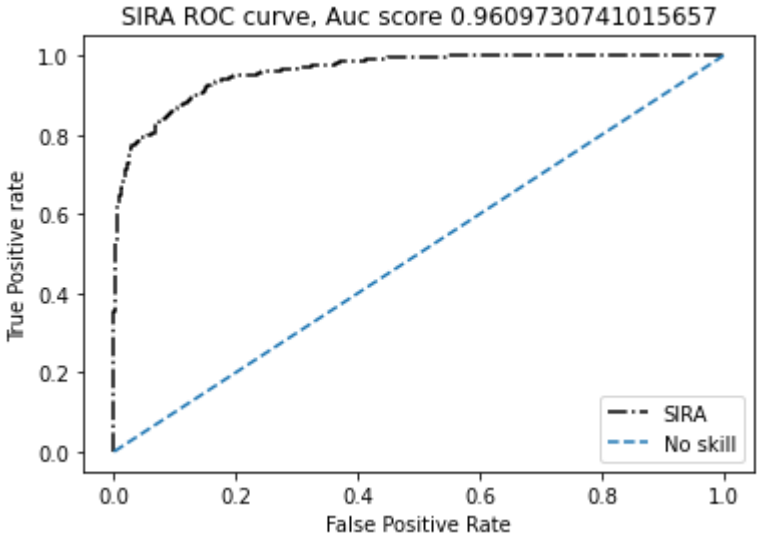

SEKER ROC curve, Auc score 0.9436649506803946
True Positive rate
False Positive Rate
SEKER
No skill
0.0
0.2
0.4
0.6
0.8
1.0

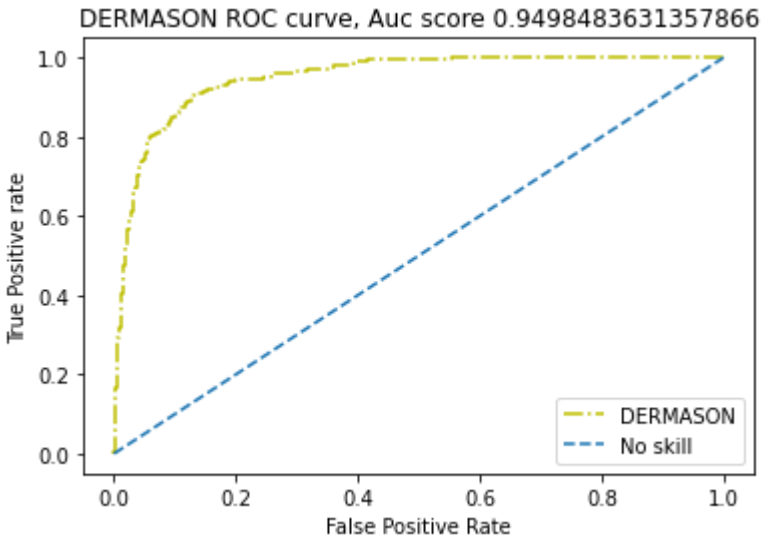

BARBUNYA ROC curve, Auc score 1.0
True Positive rate
False Positive Rate
BARBUNYA
No skill
0.0
0.2
0.4
0.6
0.8
1.0


BOMBAY ROC curve, Auc score 0.9724107642955343
True Positive rate
False Positive Rate
BOMBAY
No skill
0.0
0.2
0.4
0.6
0.8
1.0


CALI ROC curve, Auc score 0.9758205467279585
True Positive rate
False Positive Rate
CALI
No skill
0.0
0.2
0.4
0.6
0.8
1.0


HOROZ ROC curve, Auc score 0.9624486609052794
True Positive rate
False Positive Rate
HOROZ
No skill
0.0
0.2
0.4
0.6
0.8
1.0


SIRA ROC curve, Auc score 0.9609730741015657
True Positive rate
False Positive Rate
SIRA
No skill
0.0
0.2
0.4
0.6
0.8
1.0




## Comment on the output

The accuracy of the model after using Multinomial Naive bayes classifier:

BARBUNYA >> CALI >> BOMBAY >> HOROZ >> SIRA >> DERMASON >> SEKER

In [28]:
```
lr=LogisticRegression(max_iter=200,multi_class='ovr')
X_train, X_test, y_train, y_test= train_test_split(df.drop("Class",axis=1),
                                                   df["Class"], test_size=0.2,
                                                   random_state=18)

lr.fit(X_train,y_train)
y_pr=lr.predict(X_test)
print("Logistic Regression ")
print("Accuracy: ",metrics.accuracy_score(y_test, y_pr))
print("Precision Score: ",metrics.precision_score(y_test, y_pr,average='micro'))
print("Recall Score: ",metrics.recall_score(y_test, y_pr,average='micro'))
```

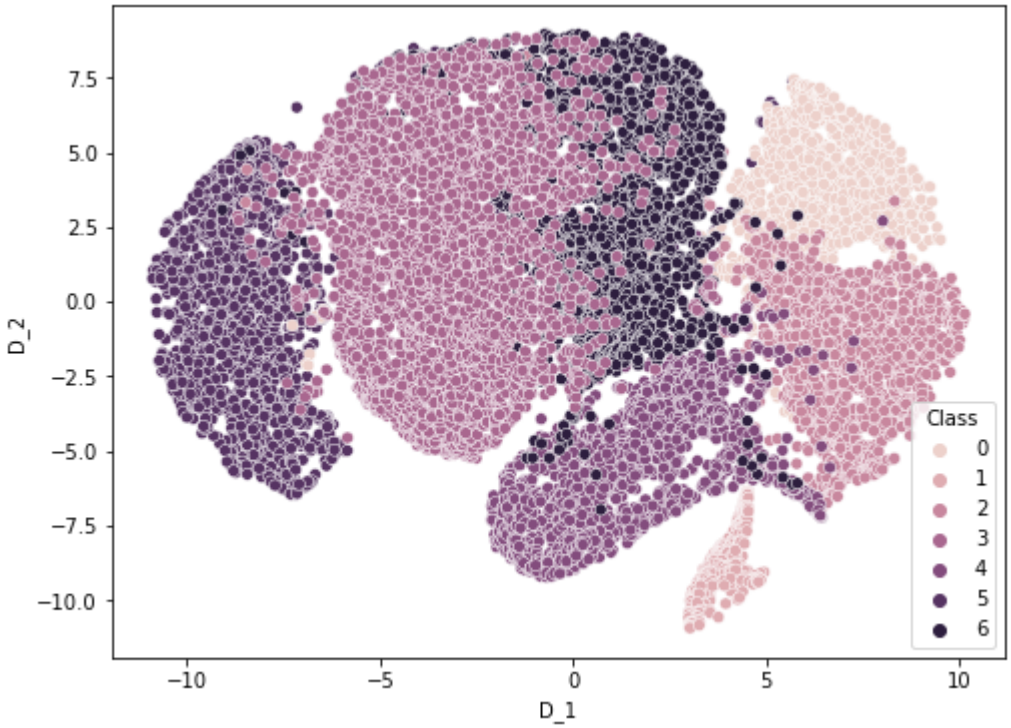
```
Logistic Regression
Accuracy:  0.868160117517444
Precision Score:  0.868160117517444
Recall Score:  0.868160117517444
```

## SEC - C ENDS HERE

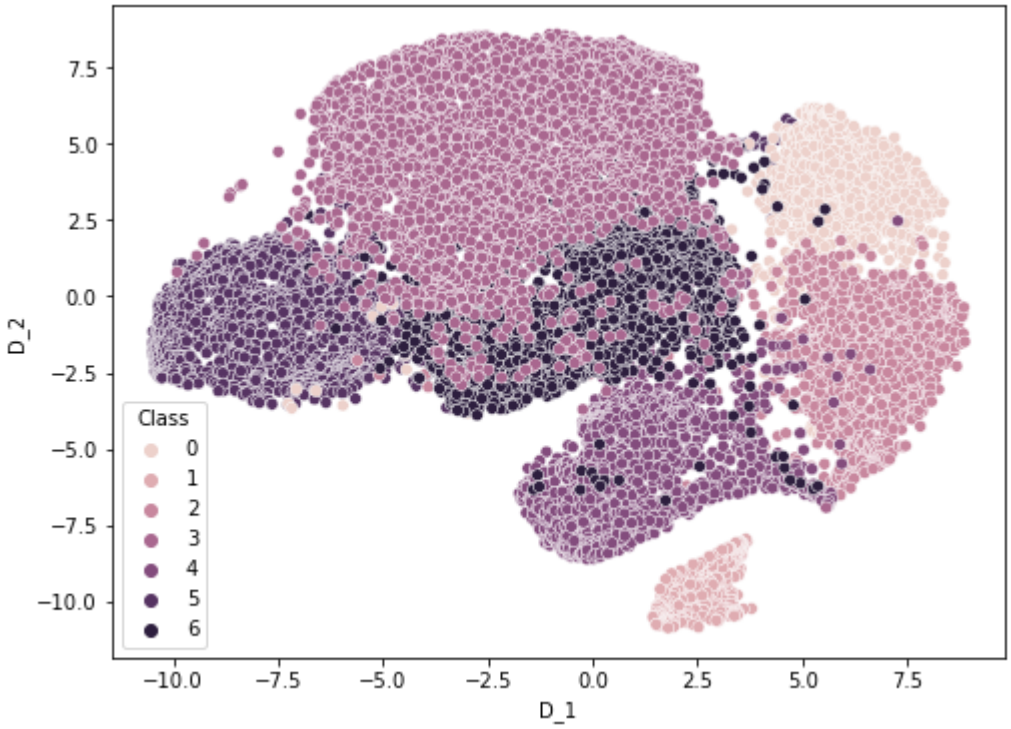
## Runnning TSNE for different preplexity for confirmation

In [ ]:
```
for m in range(30,1000,50):
    tsne_model = TSNE(n_components=2, random_state=0,perplexity=m, n_iter=300)
    tsne_data = tsne_model.fit_transform(standarized_data)
    print(tsne_data.shape)
    tsne_1=np.vstack((tsne_data.T,tar_var)).T
    tsne_1=pd.DataFrame(data=tsne_1,columns=("D_1","D_2","Class"))
    plt.figure(figsize=(8,6))
    sns.scatterplot(x="D_1",y="D_2",hue="Class",data=tsne_1)
    plt.show()
```

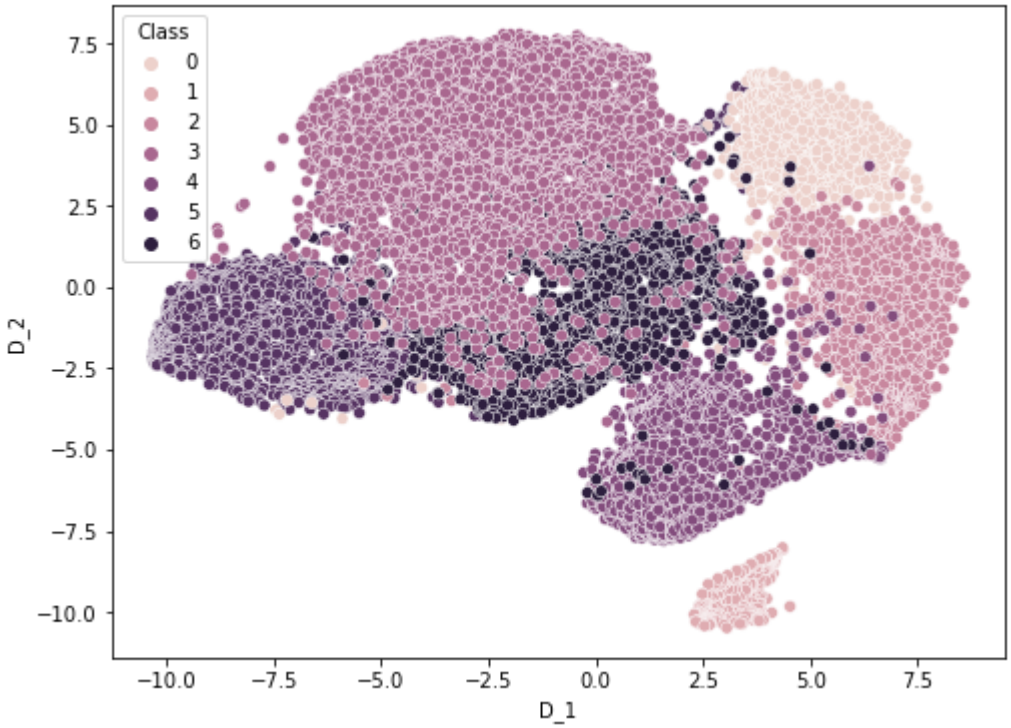
```
(13611, 2)
```



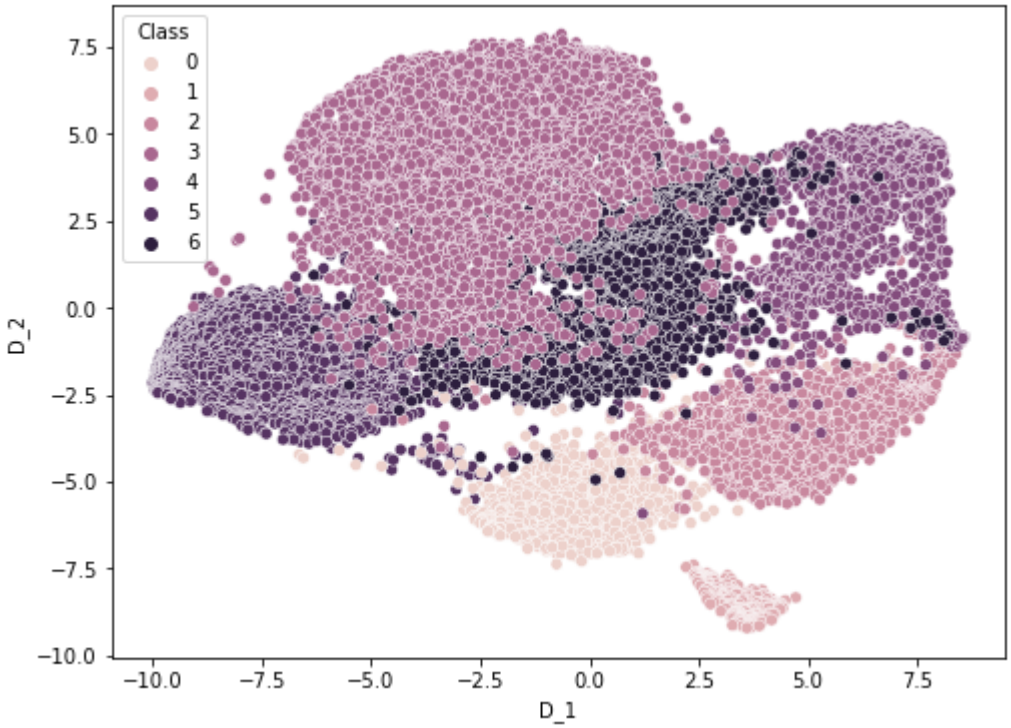

(13611, 2)



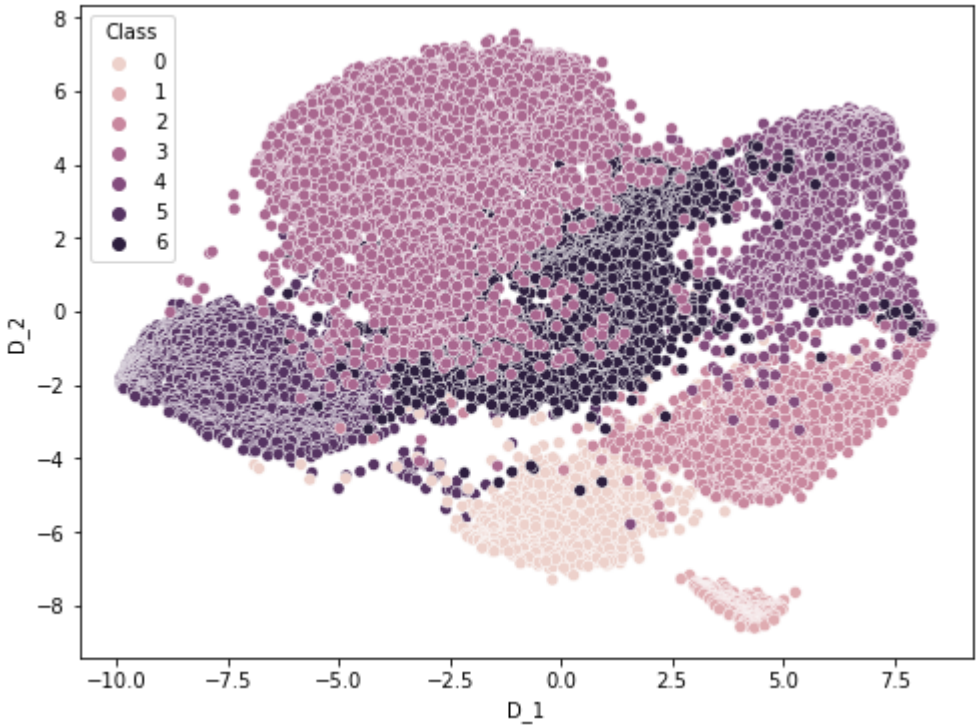

(13611, 2)



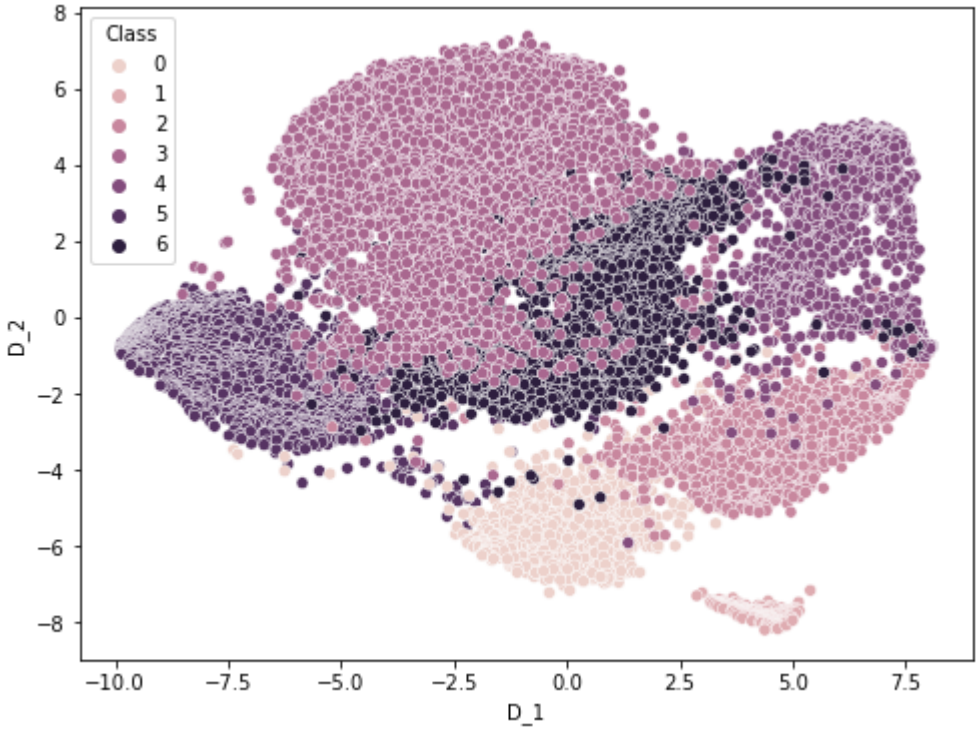

(13611, 2)



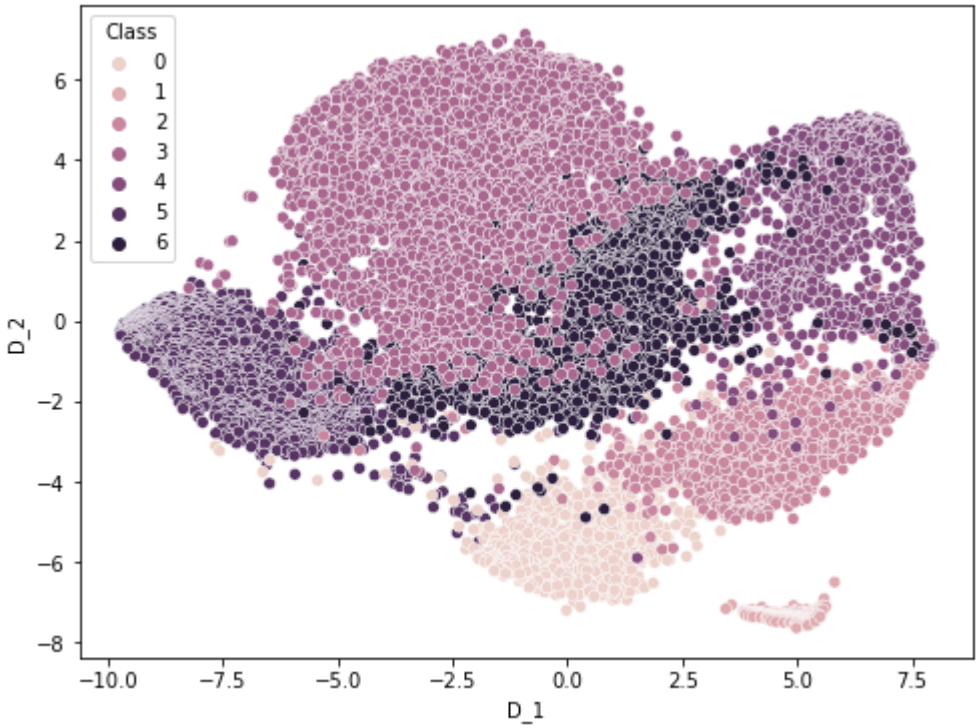

(13611, 2)



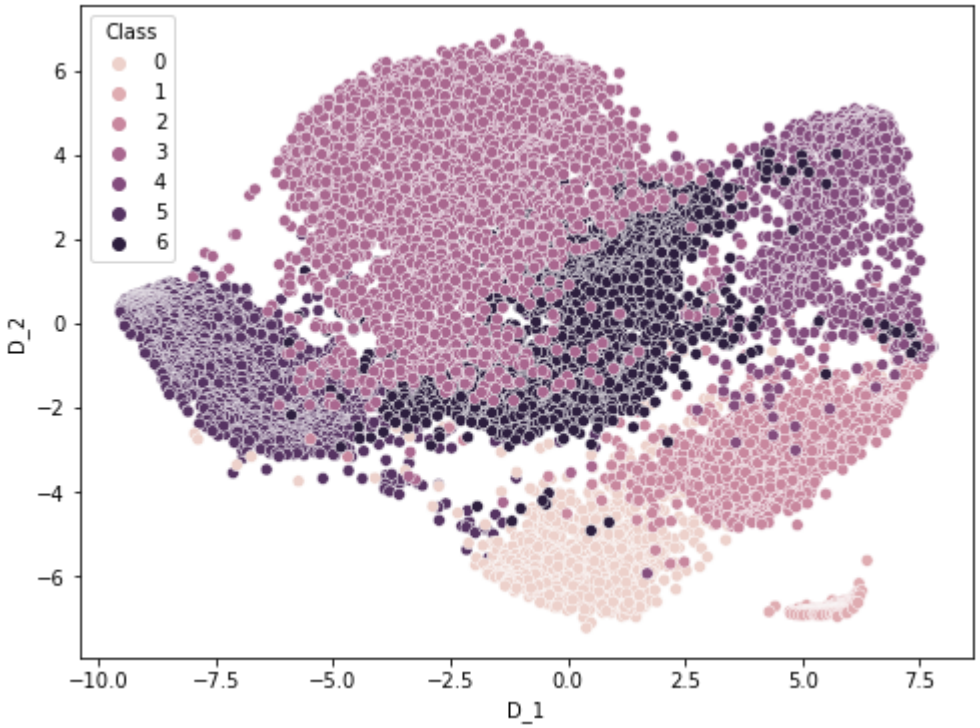

(13611, 2)



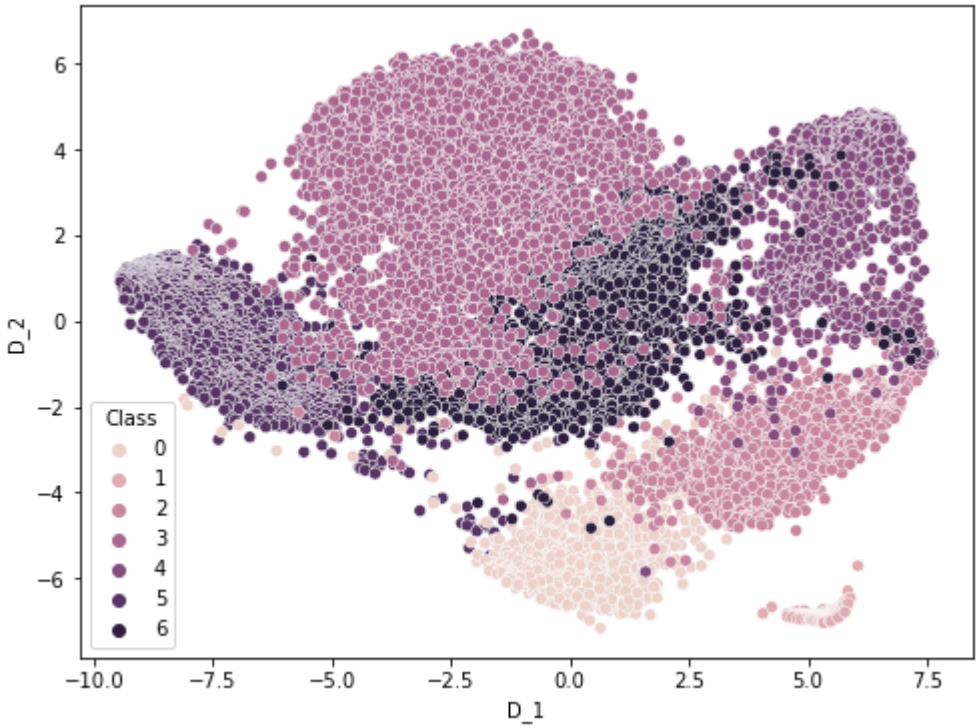

(13611, 2)



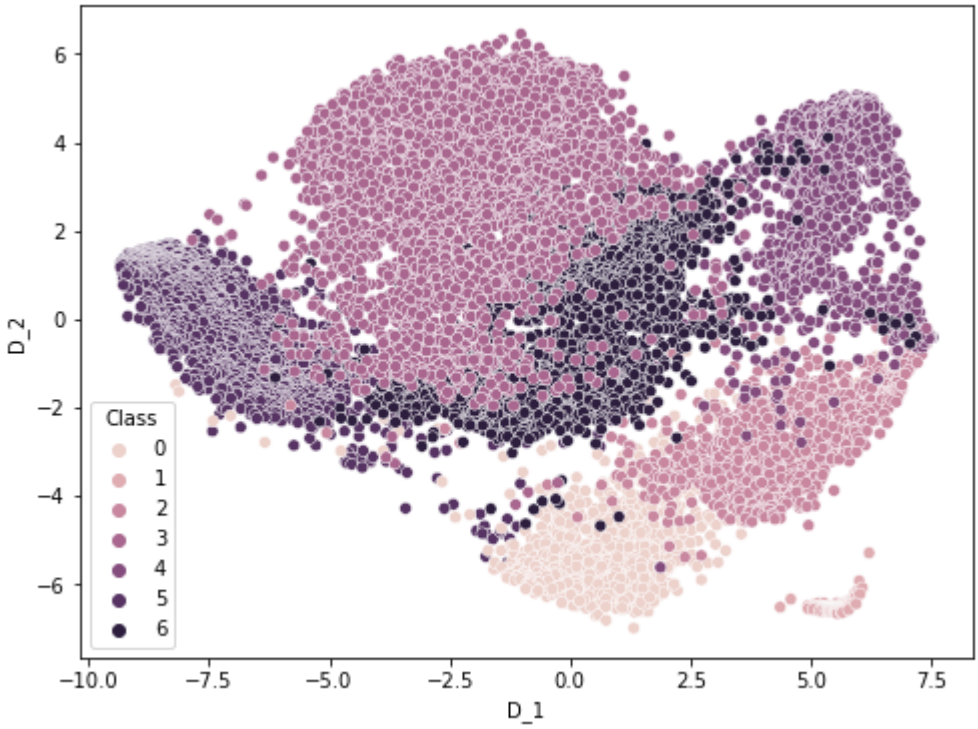

```
(13611, 2)
```




```
(13611, 2)
```




```
(13611, 2)
```




In [ ]: