Security Analytics

Assignment 2

Student Name: Chandrika Mukherjee
Student ID: 32808289
Email: cmukherj@purdue.edu

Problem 1

- **1. Principal Assumption in Naive Bayes' Model:** Input features are independent, do not affect each other. If the features are $X = (x_1, x_2, x_3, ..., x_n)$, then due to the assumptions, we can write as ,
$P(y/x_1, x_2, ..., x_n) \propto P(y)P(x_1/y)P(x_2/y)...P(x_n/y)$

  Assumption is useful : When in real-world, features don't depend on each other, then Naive Bayes works better than other classifier. It will converge quicker than other models.

- **2. k-NN do better than Logistic Regression:** When the ML model is non-linear, as Logistic regression support only linear model

- **3. 150 examples in + class, 50 examples in - class**
So, total examples = $(150 + 50) = 200$
Entropy of the class = $-[(150/200)log_2(150/200) + (50/200)log_2(50/200)]$ which is equivalent to 0.244

- **4. Number of Requests from Domain A = {2, 3, 4, 3, 4, 3, 3, 4, 5, 3, 2, 3, 4, 3, 2, 2, 3, 4, 5, 6} and Number of Requests from Domain B= {22, 23, 24, 23, 24, 23, 23, 24, 25, 23}**
total requests from A $[Sum(A)]$ = ((4*2)+(8*3)+(5*4)+(2*5)+(1*6)) = 68
total requests from B $[Sum(B)]$ = ((1*22)+(5*23)+(3*24)+(1*25)) = 234

  Mean for A $[\mu_A]$ = $Sum(A)$/(total request instances from A) = 68/20 = 3.4
  Mean for B $[\mu_B]$ = $Sum(B)$/(total request instances from B) = 234/10 = 23.4

  Variance for A $[Var(A)]$ =
  $((4 * (2 - 3.4)^2) + (8 * (3 - 3.4)^2) + (5 * (4 - 3.4)^2) + (2 * (5 - 3.4)^2) + (1 * (6 - 3.4)^2))/$
  (total request instances from A-1) = $(7.84 + 1.28 + 1.8 + 5.12 + 6.76)/(20 - 1) = 1.2$

  Variance for B $[Var(B)]$ =
  $((1 * (22 - 23.4)^2) + (5 * (23 - 23.4)^2) + (3 * (24 - 23.4)^2) + (1 * (25 - 23.4)^2))/$ (total request instances from B-1) = $(1.96 + 0.8 + 1.08 + 2.56)/(10 - 1) = 0.711$

  Standard Deviation for A $[SD_A] = \sqrt{Var(A)} = \sqrt{1.2} = 1.0954$
  Standard Deviation for B $[SD_B] = \sqrt{Var(B)} = \sqrt{0.711} = 0.843$

$$P(A) = \frac{1}{20}$$
$$P(B) = \frac{1}{10}$$

$$\begin{aligned} P(A+B) &= P(A) + P(B) - P(A \cap B) = \frac{1}{20} + \frac{1}{10} - 0 \\ &= \frac{1}{20} + \frac{1}{10} \\ &= \frac{3}{20} \end{aligned} \tag{1}$$

Prior for A, $P(A)/P(A+B) = \frac{1}{20}/\frac{3}{20} = \frac{1}{3}$

Prior for B, $P(B)/P(A+B) = \frac{1}{10}/\frac{3}{20} = \frac{2}{3}$

- **5.** Given,

$x^{(1)}$	$x^{(2)}$	y
4	7	0
-4	5	0
2	10	1
10	4	1

$P(y=0) = 2/4 = 0.5$
$P(y=1) = 2/4 = 0.5$

Using the formulas as following,

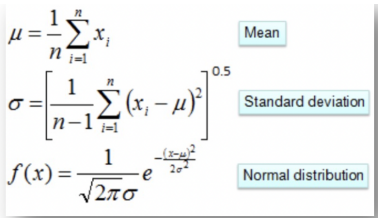


Table for $x^{(1)}$,

y	$x^{(1)}$	Mean	SD
0	4, -4	$\frac{(4-4)}{2} = 0$	$\sqrt{\frac{(4-0)^2+(-4-0)^2}{(2-1)}} = \sqrt{32}$
1	2, 10	$\frac{(10+2)}{2} = 6$	$\sqrt{\frac{(2-6)^2+(10-6)^2}{(2-1)}} = \sqrt{32}$

Table for $x^{(2)}$,

y	$x^{(2)}$	Mean	SD
0	7, 5	$\frac{(7+5)}{2} = 6$	$\sqrt{\frac{(7-6)^2+(5-6)^2}{(2-1)}} = \sqrt{2}$
1	10, 4	$\frac{(10+4)}{2} = 7$	$\sqrt{\frac{(10-7)^2+(4-7)^2}{(2-1)}} = \sqrt{18}$

According to Naive Bayes Assumption, we can write as below,

$$\begin{aligned} P(y=0|x) &= P(y=0) * [P(x^{(1)}|y=0) * P(x^{(2)}|y=0)] \\ &= 0.5 * [(\frac{1}{\sqrt{2\pi} * \sqrt{32}} * e^{\frac{-(x-0)^2}{(2*32)}}) * (\frac{1}{\sqrt{2\pi} * \sqrt{32}} * e^{\frac{-(x-6)^2}{(2*32)}})] \\ &= 0.5 * [\frac{1}{64\pi} * e^{\frac{-(x-0)^2}{(64)}} * e^{\frac{-(x-6)^2}{(64)}}] \\ &= 0.5 * [\frac{1}{64\pi} * e^{-\frac{(x^2+(x-6)^2)}{64}}] \end{aligned} \tag{2}$$

$$\begin{aligned} P(y=1|x) &= P(y=1) * [P(x^{(1)}|y=1) * P(x^{(2)}|y=1)] \\ &= 0.5 * [(\frac{1}{\sqrt{2\pi} * \sqrt{2}} * e^{\frac{-(x-6)^2}{(2*2)}}) * (\frac{1}{\sqrt{2\pi} * \sqrt{18}} * e^{\frac{-(x-7)^2}{(2*18)}})] \\ &= 0.5 * [\frac{1}{12\pi} * e^{\frac{-(x-6)^2}{(4)}} * e^{\frac{-(x-7)^2}{(36)}}] \\ &= 0.5 * [\frac{1}{12\pi} * e^{-\frac{(9*(x-6)^2+(x-7)^2)}{36}}] \end{aligned} \tag{3}$$

Problem 2

1. Decision on which attribute to consider as Root:

Total instances = 16 and out of these, 9 are "Yes" and 7 are "No"

$$\begin{aligned} H(S) &= -[\frac{9}{16} log_2(\frac{9}{16}) + \frac{7}{16} log_2(\frac{7}{16})] \\ &= 0.98847 \end{aligned} \tag{4}$$

Now, will be calculating Information gain using Color, Shape and Size.

Information Gain Using Color Attribute:
Total Yellow Instances = 13, Green Instance = 3

Out of 13 Yellow instances, for 8, we get "Yes" output and for 5, we get "No" output.

$$\begin{aligned} H(S|Yellow) &= -[\frac{8}{13}log_2(\frac{8}{13}) + \frac{5}{13}log_2(\frac{5}{13})] \\ &= 0.96070 \end{aligned} \tag{5}$$

Out of 3 Green instances, for 1, we get "Yes" output and for 2, we get "No" output.

$$\begin{aligned} H(S|Green) &= -[\frac{1}{3}log_2(\frac{1}{3}) + \frac{2}{3}log_2(\frac{2}{3})] \\ &= 0.91784 \end{aligned} \tag{6}$$

Therefore, Information gain using Color Attribute is as below,

$$\begin{aligned} InfoGain(S, Color) &= H(S) - [\frac{13}{16}H(S|Yellow) + \frac{3}{16}H(S|Green)] \\ &= 0.98847 - [\frac{13}{16} * 0.96070 + \frac{3}{16} * 0.91784] \\ &= 0.98847 - 0.952655 \\ &= 0.0358 \end{aligned} \tag{7}$$

Information Gain Using Size Attribute:
Total Small Instances = 8, Large Instance = 8

Out of 8 Small instances, for 6, we get "Yes" output and for 2, we get "No" output.

$$\begin{aligned} H(S|Small) &= -[\frac{6}{8}log_2(\frac{6}{8}) + \frac{2}{8}log_2(\frac{2}{8})] \\ &= 0.811214 \end{aligned} \tag{8}$$

Out of 8 Large instances, for 3, we get "Yes" output and for 5, we get "No" output.

$$\begin{aligned} H(S|Large) &= -[\frac{3}{8}log_2(\frac{3}{8}) + \frac{5}{8}log_2(\frac{5}{8})] \\ &= 0.95405 \end{aligned} \tag{9}$$

Therefore, Information gain using Size Attribute is as below,

$$\begin{aligned} InfoGain(S, Size) &= H(S) - [\frac{8}{16}H(S|Small) + \frac{8}{16}H(S|Large)] \\ &= 0.98847 - [\frac{8}{16} * 0.811214 + \frac{8}{16} * 0.95405] \\ &= 0.98847 - 0.882632 \\ &= 0.105838 \end{aligned} \tag{10}$$

Information Gain Using Shape Attribute:
Total Round Instances = 12, Irregular Instance = 4

Out of 12 Round instances, for 6, we get "Yes" output and for 6, we get "No" output.

$$\begin{aligned} H(S|Round) &= -[\frac{6}{12}log_2(\frac{6}{12}) + \frac{6}{12}log_2(\frac{6}{12})] \\ &= 1 \end{aligned} \tag{11}$$

Out of 4 Irregular instances, for 3, we get "Yes" output and for 1, we get "No" output.

$$\begin{aligned} H(S|Irregular) &= -[\frac{3}{4}log_2(\frac{3}{4}) + \frac{1}{4}log_2(\frac{1}{4})] \\ &= 0.811214 \end{aligned} \tag{12}$$

Therefore, Information gain using Shape Attribute is as below,

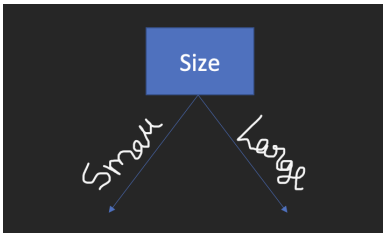
$$\begin{aligned} InfoGain(S, Shape) &= H(S) - [\frac{8}{16}H(S|Round) + \frac{8}{16}H(S|Irregular)] \\ &= 0.98847 - [\frac{12}{16} * 1 + \frac{4}{16} * 0.811214] \\ &= 0.98847 - 0.9528 \\ &= 0.03567 \end{aligned} \tag{13}$$

So, the information gain will be more if we choose **Size** as the root. (Information gain of Size is 0.105838 which is more than Information gain from Color (0.0358) and Information gain from Shape(0.03567))

Ans. We Choose Size as Root of Decision Tree.

2. Decision Tree Drawing and Corresponding Logic :

Taking the Size as Root (from the above derivation) as the Information gain is more with Root= Size



Splitting with Size attribute gives two results - Small and Large. There are total 8 Small instances. Out of 8, 6 gives "Yes" and 2 gives "No".

$$\begin{aligned} H(Small) &= -[\frac{6}{8}log_2(\frac{6}{8}) + \frac{2}{8}log_2(\frac{2}{8})] \\ &= 0.811214 \end{aligned} \tag{14}$$

There are total 8 Large instances. Out of 8, 3 gives "Yes" and 5 gives "No"

$$\begin{aligned} H(Large) &= -[\frac{3}{8}log_2(\frac{3}{8}) + \frac{5}{8}log_2(\frac{5}{8})] \\ &= 0.95405 \end{aligned} \tag{15}$$

Calculation of Information Gain with respect to Color on Small Size:
There are total 6 instances of Small Size and Yellow Color. Out of these 6, 5 give "Yes" and 1 gives "No"

$$\begin{aligned} H(Small|Yellow) &= -[\frac{5}{6}log_2(\frac{5}{6}) + \frac{1}{6}log_2(\frac{1}{6})] \\ &= 0.6497 \end{aligned} \tag{16}$$

There are total 2 instances of Small Size and Green Color. Out of these 2, 1 gives "Yes" and 1 gives "No"

$$\begin{aligned} H(Small|Green) &= -[\frac{1}{2}log_2(\frac{1}{2}) + \frac{1}{2}log_2(\frac{1}{2})] \\ &= 1 \end{aligned} \tag{17}$$

Total Small size instances are 8, out of which 6 are Small and Yellow, other 2 are Small and Green

$$\begin{aligned} InfoGain(Small, Color) &= H(Small) - [\frac{6}{8}H(Small|Yellow) + \frac{2}{8}H(Small|Green)] \\ &= 0.811214 - [\frac{6}{8} * 0.6497 + \frac{2}{8} * 1] \\ &= 0.811214 - 0.7372 \\ &= 0.074014 \end{aligned} \tag{18}$$

Calculation of Information Gain with respect to Shape on Small Size:

There are total 6 instances of Small Size and Round Shape. Out of these 6, 4 give "Yes" and 2 give "No"

$$\begin{aligned} H(Small|Round) &= -[\frac{4}{6}log_2(\frac{4}{6}) + \frac{2}{6}log_2(\frac{2}{6})] \\ &= 0.91828 \end{aligned} \tag{19}$$

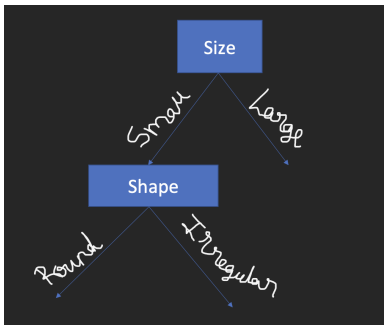
There are total 2 instances of Small Size and Irregular Shape. 2 of them give "Yes".

$$\begin{aligned} H(Small|Irregular) &= -[\frac{2}{2}log_2(\frac{2}{2})] \\ &= 0 \end{aligned} \tag{20}$$

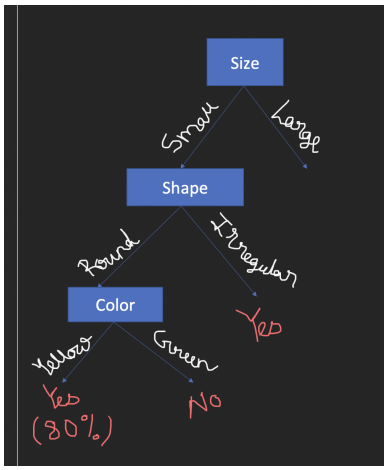
Total small size instances are 8, out of which 6 are Small and Round, other 2 are Small and Irregular

$$\begin{aligned} InfoGain(Small, Shape) &= H(Small) - [\frac{6}{8}H(Small|Round) + \frac{2}{8}H(Small|Irregular)] \\ &= 0.811214 - [\frac{6}{8} * 0.91828 + \frac{2}{8} * 0] \\ &= 0.811214 - 0.68871 \\ &= 0.122504 \end{aligned} \tag{21}$$

Therefore, after splitting using Size, in the Small branch, we should split using Shape as Information Gain by Shape is 0.122504 that is higher than Information Gain by Color (0.074014).



Now, After Splitting Small branch with Shape, Consequent Irregular branch will have only one outcome ("Yes"), Entropy is also Zero. But the Entropy of Round branch is more than zero (0.91828). Therefore, Need to split Round Branch with Color. After Splitting with Color, Green Branch has only one outcome "No" as in the training data, only one instance is present with {Size=Small, Shape= Round, Color= Green} which has output of "No". But there are total 5 instances of {Size=Small, Shape= Round, Color= Yellow}, out of these 5, 4 give "Yes" as output and 1 gives "No" as output. As we don't have any other feature, if we consider majority, Yellow branch will give "Yes" as output.



Calculation of Information Gain with respect to Color on Large Size:

There are total 7 instances of Large Size and Yellow Color. Out of these 7, 3 give "Yes" and 4 gives "No"

$$\begin{aligned} H(Large|Yellow) &= -[\frac{3}{7}log_2(\frac{3}{7}) + \frac{4}{7}log_2(\frac{4}{7})] \\ &= 0.98518 \end{aligned} \tag{22}$$

There are total 1 instance of Large Size and Green Color and it gives output as "No"

$$\begin{aligned} H(Large|Green) &= -[\frac{1}{1}log_2(\frac{1}{1})] \\ &= 0 \end{aligned} \tag{23}$$

Total Large size instances are 8, out of which 7 are Large and Yellow, other 1 is Large and Green

$$\begin{aligned} InfoGain(Large, Color) &= H(Large) - [\frac{7}{8}H(Large|Yellow) + \frac{1}{8}H(Large|Green)] \\ &= 0.95405 - [\frac{7}{8} * 0.98518 + \frac{1}{8} * 0] \\ &= 0.95405 - 0.86203 \\ &= 0.09202 \end{aligned} \tag{24}$$

Calculation of Information Gain with respect to Shape on Large Size:

There are total 6 instances of Large Size and Round Shape. Out of these 6, 2 give "Yes" and 4 give "No"

$$\begin{aligned} H(Large|Round) &= -[\frac{2}{6}log_2(\frac{2}{6}) + \frac{4}{6}log_2(\frac{4}{6})] \\ &= 0.91828 \end{aligned} \tag{25}$$

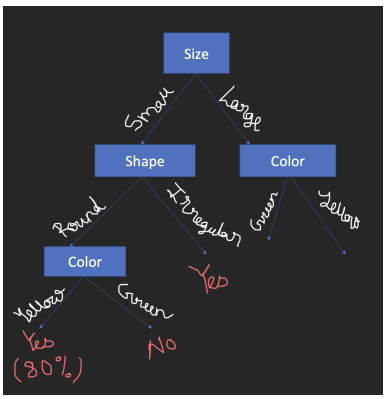
There are total 2 instances of Large Size and Irregular Shape. Out of 2, 1 gives "Yes" and 1 gives "No".

$$\begin{aligned} H(Large|Irregular) &= -[\frac{1}{2}log_2(\frac{1}{2}) + \frac{1}{2}log_2(\frac{1}{2})] \\ &= 1 \end{aligned} \tag{26}$$

Total large size instances are 8, out of which 6 are Large and Round, other 2 are Large and Irregular

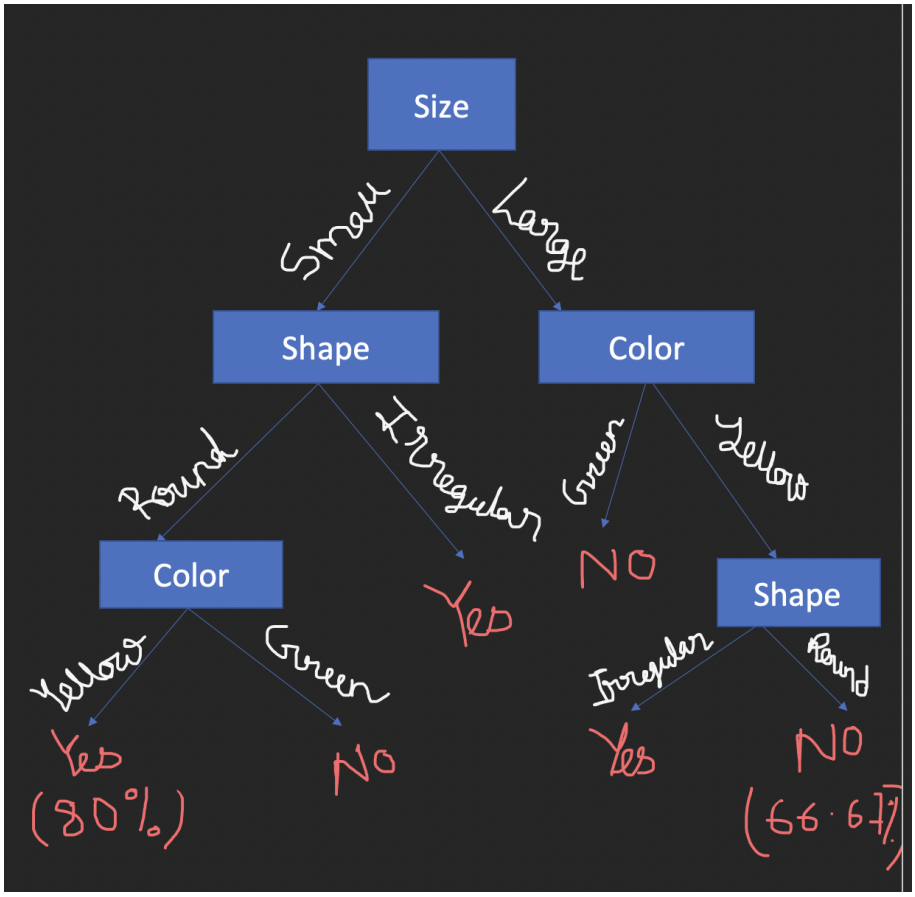
$$\begin{aligned} InfoGain(Large, Shape) &= H(Large) - [\frac{6}{8}H(Large|Round) + \frac{2}{8}H(Large|Irregular)] \\ &= 0.95405 - [\frac{6}{8} * 0.91828 + \frac{2}{8} * 1] \\ &= 0.95405 - 0.93871 \\ &= 0.01534 \end{aligned} \tag{27}$$

Therefore, after splitting using Size, in the Large branch, we should split using Color as Information Gain by Color is 0.09202 that is higher than Information Gain by Shape (0.01534).



Now, After Splitting Large branch with Color, Consequent Green branch will have only one outcome ("No"), Entropy is also Zero. But the Entropy of Yellow branch is more than zero (0.98518). Therefore, Need to split Yellow Branch with Shape. After Splitting with Shape, Irregular Branch has only one outcome "Yes" as in the training data, only one instance is present with {Size=Large, Color= Yellow, Shape= Irregular } which has output of "Yes". But there are total 6 instances of {Size=Large, Color= Yellow, Shape= Round}, out of these 6, 2 give "Yes" as output and 4 give "No" as output. As we don't have any other feature, if we consider majority, Round branch will give "No" as output.

This is the final Decision Tree.



3. If Features have Numerical Values and those are treated as Discrete, and we proceed as Categorical Decision Tree Algorithm :

When the decision tree is used to classify unseen data, problems will arise.
New unseen data may not be present in the sample training data set. So, the branch corresponding to the unseen value will not be present in the decision tree, therefore, from that point, decision process will stop.
For example, In the given data set of edible mushroom, if there was another feature "Height" which had numerical values. If we treated each value as discrete, then we could build the decision tree. Suppose,The heights present in the data were = {10,12,20,21,24,11,56,34,21,22,10,12,13,24,55,22}. Now, new unseen data has height = 28, but we don't have any branch corresponding to height 28, then we can't conclude on the output. Even if unseen data is present in training sample, treating each numerical value as discrete, make the decision tree more complex.

Problem3

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import random
random.seed(0)
np.random.seed(0)
```

In [2]:
```
columns = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI'
           , 'DiabetesPedigreeFunction', 'Age','Target']
```

In [3]:
```
# Read the data
pima = pd.read_csv('Pima.csv',names=columns)
```

In [4]:
```
pima.head()
```

Out[4]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Target
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

In [5]:
```
# Data information (columns and rows)
pima.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   Pregnancies              768 non-null    int64  
 1   Glucose                  768 non-null    int64  
 2   BloodPressure            768 non-null    int64  
 3   SkinThickness            768 non-null    int64  
 4   Insulin                  768 non-null    int64  
 5   BMI                      768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                      768 non-null    int64  
 8   Target                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

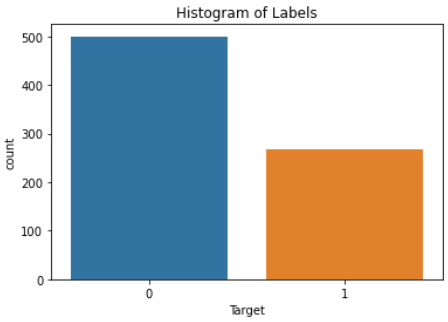
In [6]:
```
# Data Statistics
pima.describe()
```

Out[6]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Target
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

In [7]:
```
# Histogram of labels
sns.countplot(x='Target',data=pima)
plt.title("Histogram of Labels")
```

Out[7]:
```
Text(0.5, 1.0, 'Histogram of Labels')
```



In [8]:
```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
```

In [9]:
```
X=pima.iloc[:,:8]
y = pima.iloc[:,8:]
accuracy=[]
error=[]
```

In [10]:
```
# 2. Splitting the data (training (80%) and test (20%))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

In [11]:
```
# K Value Between 1 and 15
for k in range(1,16):
    print("k = ",k)
    knn = KNeighborsClassifier(n_neighbors = k)
    scores = cross_val_score(knn, X_train, y_train.values.ravel(), cv=5, scoring='accuracy') # cv=5 -> no of folds
    print(scores)
    acc = scores.mean()
    print("mean accuracy for k = ",k,"is : ",acc)
    accuracy.append(acc)
    print("********************")
    error.append(1-acc)
```

```
k =  1
[0.6504065  0.68292683 0.69105691 0.67479675 0.63114754]
mean accuracy for k =  1 is :  0.6660669065707051
********************
k =  2
[0.65853659 0.69918699 0.72357724 0.69918699 0.69672131]
mean accuracy for k =  2 is :  0.6954418232706917
********************
k =  3
[0.67479675 0.68292683 0.68292683 0.64227642 0.69672131]
mean accuracy for k =  3 is :  0.6759296281487407
********************
k =  4
[0.67479675 0.70731707 0.65853659 0.69105691 0.72131148]
mean accuracy for k =  4 is :  0.6906037584966013
********************
k =  5
[0.70731707 0.69918699 0.67479675 0.69105691 0.7704918 ]
mean accuracy for k =  5 is :  0.7085699053711847
********************
k =  6
[0.69105691 0.73170732 0.7398374  0.69105691 0.7704918 ]
mean accuracy for k =  6 is :  0.7248300679728109
********************
k =  7
[0.72357724 0.72357724 0.70731707 0.68292683 0.76229508]
mean accuracy for k =  7 is :  0.7199386911901906
********************
k =  8
[0.69105691 0.72357724 0.74796748 0.70731707 0.72131148]
mean accuracy for k =  8 is :  0.7182460349193656
********************
k =  9
[0.72357724 0.71544715 0.69105691 0.71544715 0.7295082 ]
mean accuracy for k =  9 is :  0.7150073304011728
********************
k =  10
[0.69105691 0.73170732 0.73170732 0.69918699 0.74590164]
mean accuracy for k =  10 is :  0.7199120351859256
********************
k =  11
[0.69105691 0.73170732 0.75609756 0.69105691 0.75409836]
mean accuracy for k =  11 is :  0.7248034119685458
********************
k =  12
[0.66666667 0.74796748 0.75609756 0.69105691 0.7295082 ]
mean accuracy for k =  12 is :  0.7182593629214982
********************
k =  13
[0.68292683 0.72357724 0.7398374  0.69105691 0.73770492]
mean accuracy for k =  13 is :  0.7150206584033054
********************
k =  14
[0.68292683 0.7398374  0.7398374  0.71544715 0.74590164]
mean accuracy for k =  14 is :  0.7247900839664134
********************
k =  15
[0.68292683 0.69918699 0.74796748 0.70731707 0.74590164]
mean accuracy for k =  15 is :  0.7166600026656005
********************
```

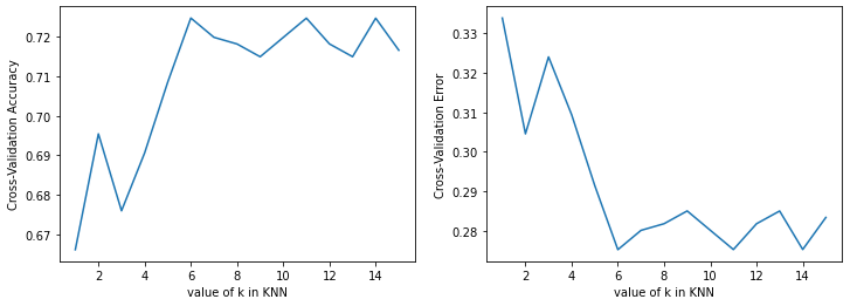
In [12]:
```
print(accuracy) # accuracy for for K=1 to 16
```

```
[0.6660669065707051, 0.6954418232706917, 0.6759296281487407, 0.6906037584966013, 0.7085699053711847, 0.7248300679728109, 0.7199386911901906, 0.7182460349193656, 0.7150073304011728, 0.7199120351859256, 0.7248034119685458, 0.7182593629214982, 0.7150206584033054, 0.7247900839664134, 0.7166600026656005]
```

In [13]:
```
# cross validation error for K=1 to 16
print(error)
```

```
[0.3339330934292949, 0.3045581767293083, 0.3240703718512593, 0.30939624150339873, 0.2914300946288153, 0.2751699320271891, 0.2800613088098094, 0.2817539650806344, 0.2849926695988272, 0.2800879648140744, 0.2751965880314542, 0.28174063707850183, 0.2849793415966946, 0.2752099160335866, 0.2833399973343995]
```

In [15]:
```
plt.figure(figsize=(12,4))
k_range = range(1,16)
plt.subplot(1,2,1)
plt.plot(k_range,accuracy)
plt.xlabel('value of k in KNN')
plt.ylabel('Cross-Validation Accuracy')
plt.subplot(1,2,2)
plt.plot(k_range,error)
plt.xlabel('value of k in KNN')
plt.ylabel('Cross-Validation Error')
plt.show()
```



3. Accuracy max for K=6, for all other K values, accuracy is less. So, I will choose K=6

In [16]:
```
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train,y_train.values.ravel())
y_pred = knn.predict(X_test)
y_true = y_test.to_numpy().flatten()
total_error=0
total_accuracy=0
for j in range(len(y_pred)):
    if y_pred[j]!=y_true[j]:
        total_error+=1
    else:
        total_accuracy+=1
print("test error ",total_error/len(y_pred))
print("accuracy ",total_accuracy/len(y_pred))
```

```
test error  0.22077922077922077
accuracy  0.7792207792207793
```

In [17]:
```
X_train_std = (X_train-X_train.mean())/X_train.std()
X_test_std = (X_test-X_test.mean())/X_test.std()
```

In [18]:
```
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train_std,y_train.values.ravel())
y_pred_std = knn.predict(X_test_std)
y_true_std = y_test.to_numpy().flatten()
total_error_std=0
total_correct_std=0
for j in range(len(y_pred_std)):
    if y_pred_std[j]!=y_true_std[j]:
        total_error_std+=1
    else:
        total_correct_std+=1
print("test error after standardization ",total_error_std/len(y_pred_std))
print("accuracy after standardization", total_correct_std/len(y_pred_std))
```

```
test error after standardization  0.2012987012987013
accuracy after standardization 0.7987012987012987
```

Yes, centralization and standarization impact the accuracy - because, if the value of different features are very different, then features with larger value will dominate while computing distance, hence will impact the outcome of KNN. Centralization and standardization solve this issue. Therefore, the outcome becomes more reliable.

In [ ]:

Problem 4

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import sys
import random
random.seed(0)
np.random.seed(0)
```

In [2]:
```
column_names = ['sepal length','sepal width','petal length','petal width', 'class']
```

In [3]:
```
# read iris dataset
iris = pd.read_csv('iris.data',names=column_names,index_col=False)
```

In [4]:
```
iris.head()
```

Out[4]:

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [5]:
```
iris_df_new = pd.DataFrame() # creating new dataframe
```

In [6]:
```
# this function assigns values
def get_class(x):
    if x=='Iris-setosa': #  Iris-setosa =0
        return 0
    elif(x=='Iris-versicolor'): # Iris-versicolor =1
        return 1
    else:
        return 2 # Iris Virginica =3
```

In [7]:
```
# Assgning x1 and x2
iris_df_new['x1']=iris['sepal length']/iris['sepal width']
iris_df_new['x2']=iris['petal length']/iris['petal width']
iris_df_new['class']=iris['class']
iris_df_new['class_enc']=iris_df_new['class'].apply(lambda x: get_class(x))
```

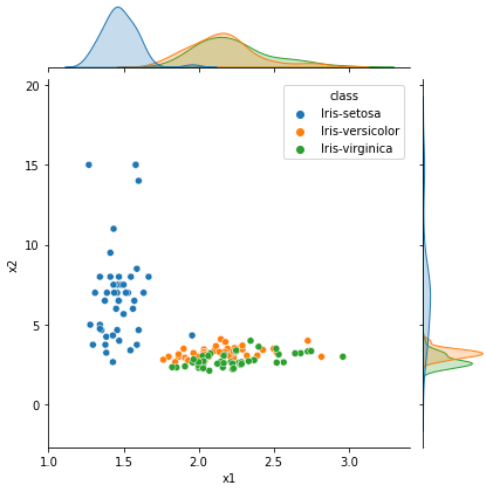
In [8]:
```
iris_df_new.head()
```

Out[8]:

	x1	x2	class	class_enc
0	1.457143	7.0	Iris-setosa	0
1	1.633333	7.0	Iris-setosa	0
2	1.468750	6.5	Iris-setosa	0
3	1.483871	7.5	Iris-setosa	0
4	1.388889	7.0	Iris-setosa	0

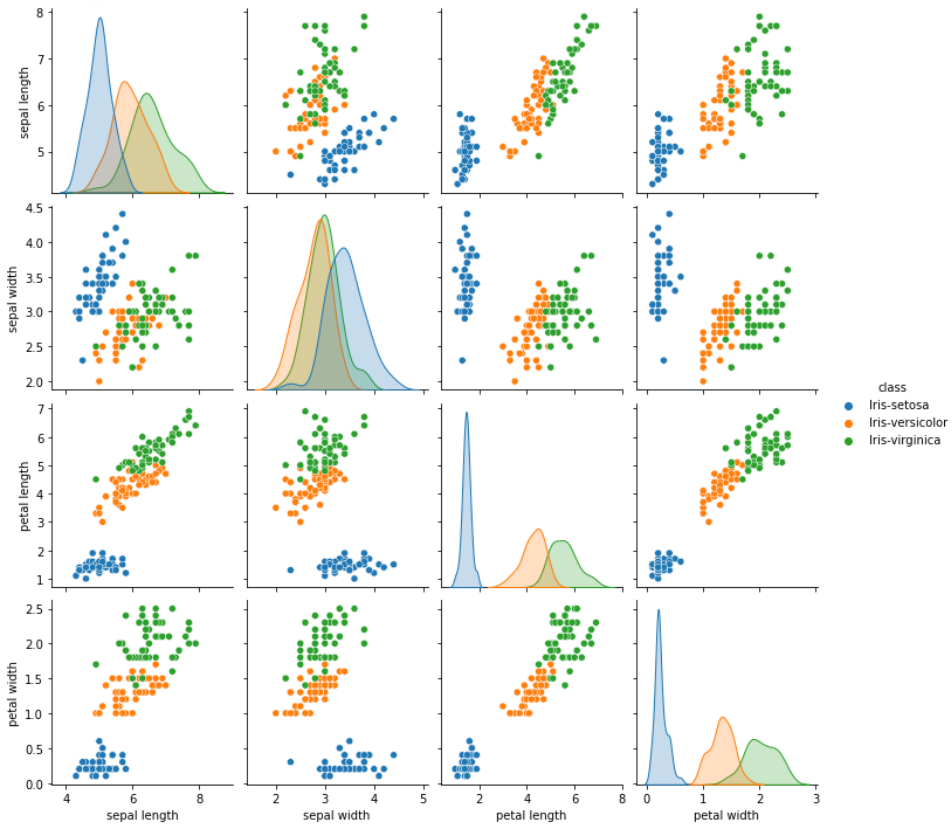
In [9]:
```
# showing the clusters
plt.figure(figsize=(12,8))
data = iris_df_new.drop('class_enc',axis=1)
sns.jointplot(x='x1',y='x2',data=data,hue='class')
```

Out[9]:
```
<seaborn.axisgrid.JointGrid at 0x7fb85005bcd0>

<Figure size 864x576 with 0 Axes>
```



In [10]:
```
sns.pairplot(data=iris,hue='class')
```

Out[10]:
```
<seaborn.axisgrid.PairGrid at 0x7fb85005be90>
```



2) KMeans++ Source Code Documentation:

1. **Using Euclidean Distance for calculating distance between data points (get_distance function)**
2. **initialize_kmeans_plus_plus -> Initializes Centroids (takes "K" as argument) and returns initial K centroids**

   We initialize a list of centroids

   centroids=[]

   a. First we randomly select a point and append to the centroids list b. Then for rest (K-1) times,

   ```
   i. We loop through all the points
       .> Then determine minimum distance of the point from all current centroids in the centroids list
   ii. Among those minimum distances, we now select the distance with max value
        and select the corresponding data point as new centroid.
   iv. Append the new centroid data point to centroids list
   ```

3. **KMeans Plus Plus Algorithm:**

   kmeans_plus_plus -> (takes current centroids, "K" value), (returns new centroids,point to cluster centroids mapping)

   a. (initialization)

   ```
   cent_map=[] stores centroids and corresponding assigned points mapping
   sums_x1=[] # stores sum of x1 feature of all points within a cluster
   sums_x2=[] # stores sum of x2 feature of all points within a cluster
   ```

   b. For each of the points,

   ```
   i. we calculate the distance from all the current centroids.
   ii. we map the data point to the centroid with which it has minimum distance
   ```

   c. For each of the cluster, we calculate the new centroids by

   ```
   new_cx1[k] = sums_x1[k]/length(cent_map[k])
   new_cx2[k] = sums_x2[k]/length(cent_map[k])
   append this values to new centroids list
   ```

   d. Returns the new centroids list and mapping of points to centroids.

In [11]:
```
# Euclidean Distance - to calculate distance
def get_distance(x1,x2,y1,y2):
    d1=(y1-x1)**2
    d2=(y2-x2)**2
    return (d1+d2)
```

In [12]:
```
## Initialize the centroids function
num_rows = iris_df_new.shape[0]
def initialize_kmeans_plus_plus(k):
    centroids_x1=[]
    centroids_x2=[]
    # first point is selected randomly
    rand_num = np.random.randint(num_rows-1)
    first_centroid_x1 = iris_df_new['x1'].iloc[rand_num]
    first_centroid_x2 = iris_df_new['x2'].iloc[rand_num]
    centroids_x1.append(first_centroid_x1)
    centroids_x2.append(first_centroid_x2)

    for k_id in range(k-1):
        dist=[]
        # for each of the point
        for ind in range(num_rows):
            x1=iris_df_new['x1'].iloc[ind]
            x2=iris_df_new['x2'].iloc[ind]
            min_dist = sys.maxsize
            # we determine the minimum distance from current all centroids
            for cd in range(len(centroids_x1)):
                y1=centroids_x1[cd]
                y2=centroids_x2[cd]
                distance = get_distance(x1,x2,y1,y2)
                min_dist = min(min_dist,distance)
            dist.append(min_dist)
        # then select the data point which is at maximum distance,
        # and assign that data point as new centroid
        mx_ind = np.argmax(np.array(dist))
        centroids_x1.append(iris_df_new['x1'].iloc[mx_ind])
        centroids_x2.append(iris_df_new['x2'].iloc[mx_ind])

    return centroids_x1,centroids_x2
```

In [13]:
```
## Kmean++ algorithm
def kmeans_plus_plus(k,centroids_x1,centroids_x2):
    # number of datapoints
    num_rows = iris_df_new.shape[0]
    cent_map=[] # stores centroids and corresponding assigned points mapping
    sums_x1=[] # stores sum of x1 feature of all points within a cluster
    sums_x2=[] # stores sum of x2 feature of all points within a cluster
    for ind in range(k):
        cent_map.append([])
        sums_x1.append(0)
        sums_x2.append(0)
    # for each data point
    for ind in range(num_rows):
        x1=iris_df_new['x1'].iloc[ind]
        x2=iris_df_new['x2'].iloc[ind]
        min_dist = sys.maxsize
        mid=-1
        # we calculate the distances between point and all centroids
        # point will be mapped to the centroid with minimum distance from it
        for cd in range(len(centroids_x1)):
            y1=centroids_x1[cd]
            y2=centroids_x2[cd]
            distance = get_distance(x1,x2,y1,y2)
            if mid==-1:
                mid=cd
                min_dist = distance
            else:
                if(distance<min_dist):
                    mid=cd
                    min_dist = distance
        cent_map[mid].append(ind)
    # calculating new centroids
    for ind in range(len(cent_map)):
        for cd in range(len(cent_map[ind])):
            index = cent_map[ind][cd]
            sums_x1[ind]+=iris_df_new['x1'].iloc[index]
            sums_x2[ind]+=iris_df_new['x2'].iloc[index]

    for ind in range(len(cent_map)):
        length = len(cent_map[ind])
        #print(length)
        if(length>0):
            sums_x1[ind]=sums_x1[ind]/float(length)
            sums_x2[ind]=sums_x2[ind]/float(length)

    centroids_x1=sums_x1
    centroids_x2=sums_x2
    # returning new centroids, mapping of points to corresponding centroids
    return centroids_x1,centroids_x2,cent_map
```

In [14]:
```
# 3. K=1 to 5 and 50 iterations for each
maps_of_points=[]
final_centroids=[]
initial_centroids=[]
for k_value in range(1,6):
    centroids_x1,centroids_x2=initialize_kmeans_plus_plus(k_value)
    first_cent=[]
    first_cent.append(centroids_x1)
    first_cent.append(centroids_x2)
    initial_centroids.append(first_cent)
    final_map=[]
    for turn in range(50):
        centroids_x1,centroids_x2,cent_map = kmeans_plus_plus(k_value,centroids_x1,centroids_x2)
        final_map=cent_map
    maps_of_points.append(final_map)
    centroids =[]
    centroids.append(centroids_x1)
    centroids.append(centroids_x2)
    final_centroids.append(centroids)
```

In [15]:
```
for k in range(len(final_centroids)):
    print("centroids for k_value = ",k+1)
    print("Initial Centroids")
    for t in range(len(initial_centroids[k][0])):
        print(initial_centroids[k][0][t],initial_centroids[k][1][t])
    print("\nFinal Centroids")
    for t in range(len(final_centroids[k][0])):
        print(final_centroids[k][0][t],final_centroids[k][1][t])
    print("\n\n*******\n\n")
```

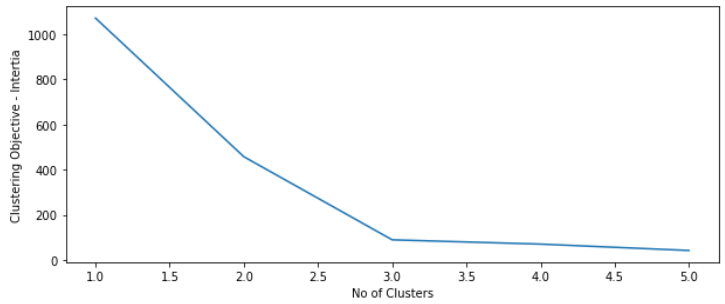
```
centroids for k_value =  1
Initial Centroids
1.4374999999999998 6.999999999999999

Final Centroids
1.9551444308694617 4.367166423691872


*******


centroids for k_value =  2
Initial Centroids
2.0263157894736845 3.0454545454545454
1.2682926829268295 15.0

Final Centroids
1.9777366323385024 3.920104640236229
1.4936180294304975 13.5


*******


centroids for k_value =  3
Initial Centroids
2.148148148148148 4.1
1.2682926829268295 15.0
1.411764705882353 9.499999999999998

Final Centroids
2.0928950450949397 3.1673129041992576
1.5220456333595596 14.8
1.4788141804347774 7.3678160919540225


*******


centroids for k_value =  4
Initial Centroids
2.1724137931034484 3.1111111111111107
1.2682926829268295 15.0
1.411764705882353 9.499999999999998
1.5714285714285714 6.5

Final Centroids
2.1066310718427834 3.1351604990097712
1.5220456333595596 14.8
1.4838758832268053 8.625
1.4623690090317294 6.724637681159421


*******


centroids for k_value =  5
Initial Centroids
1.5806451612903227 15.0
2.0714285714285716 2.125
1.5882352941176472 8.5
1.2777777777777777 5.0
1.4333333333333333 11.0

Final Centroids
1.5220456333595596 14.8
2.172285144648706 2.984749635537806
1.48232743051511 7.211538461538462
1.5910182803613162 4.3882352941176475
1.4225490196078432 10.25


*******
```

In [16]:
```
# calculating inertia (sum of square of distance of points from their cluster centroids)
acc=[]
for u in range(len(maps_of_points)):
    fmap=maps_of_points[u]
    centroids = final_centroids[u]
    sum_dist=0
    for t in range(len(fmap)):
        pts = fmap[t]
        mn_dist=0
        for y in range(len(pts)):
            dist = get_distance(centroids[0][t],centroids[1][t],
                                iris_df_new['x1'].iloc[pts[y]],iris_df_new['x2'].iloc[pts[y]])
            mn_dist+=dist
        sum_dist+=mn_dist
    acc.append(sum_dist)
    print("mean =", sum_dist, "for K = ",u+1)
```

```
mean = 1071.228686156884 for K =  1
mean = 457.22352154601197 for K =  2
mean = 89.17750273047949 for K =  3
mean = 70.1996474420087 for K =  4
mean = 42.03957396126128 for K =  5
```

In [17]:
```
plt.figure(figsize=(10,4))
index=range(1,6)
sns.lineplot(x=index, y=acc)
plt.xlabel('No of Clusters')
plt.ylabel('Clustering Objective - Intertia')
```

Out[17]:
```
Text(0, 0.5, 'Clustering Objective - Intertia')
```



4) From K =1 to 3, the inertia (sum of square of distance of points from their cluster centroids) decreases highly, then the decrease rate is very less. So, I will choose K=3

In [18]:
```
#chosen cluster = 3
centroids_x13,centroids_x23=initialize_kmeans_plus_plus(3)
print("initial chosen centroids with k=3 ")
for u in range(len(centroids_x13)):
        print(centroids_x13[u],centroids_x23[u])
```

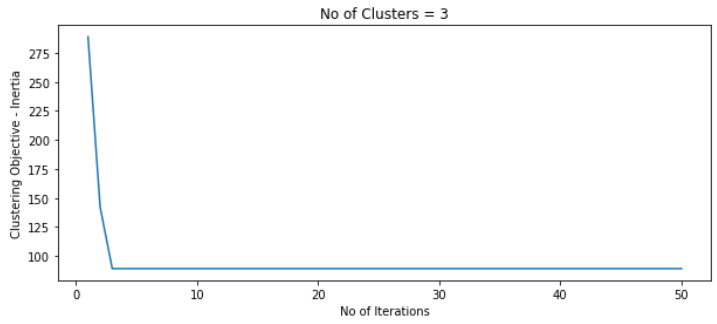
```
initial chosen centroids with k=3
1.3783783783783783 3.75
1.5806451612903227 15.0
1.411764705882353 9.499999999999998
```

In [19]:
```
final_map3=[]
cent_map3=[]
accuracy3=[]
for u in range(3):
    cent_map3.append([])
num_rows = iris_df_new.shape[0]
# initial assignment of points to initial cluster centers
for ind in range(num_rows):
        x1=iris_df_new['x1'].iloc[ind]
        x2=iris_df_new['x2'].iloc[ind]
        min_dist = sys.maxsize
        mid=-1
        for cd in range(len(centroids_x13)):
            y1=centroids_x13[cd]
            y2=centroids_x23[cd]
            distance = get_distance(x1,x2,y1,y2)
            if mid==-1:
                mid=cd
                min_dist = distance
            else:
                if(distance<min_dist):
                    mid=cd
                    min_dist = distance
        cent_map3[mid].append(ind)

for turn in range(50):
    #print("iteration = ",turn+1)
    sum_dist=0
    for u in range(len(cent_map3)):
        pts=cent_map3[u]
        mn_dist=0
        for j in range(len(pts)):
            x1=iris_df_new['x1'].iloc[pts[j]]
            x2=iris_df_new['x2'].iloc[pts[j]]
            dist = get_distance(centroids_x13[u],centroids_x23[u],x1,x2)
            mn_dist+=dist
        sum_dist+=mn_dist
    #print(sum_dist,"\n\n")
    accuracy3.append(sum_dist)
    centroids_x13,centroids_x23,cent_map3 = kmeans_plus_plus(3,centroids_x13,centroids_x23)
    final_map3=cent_map3
print("objective function (accuracy) changes for 50 iterations and K =3")
print(accuracy3)
```

```
objective function (accuracy) changes for 50 iterations and K =3
[288.86800513774403, 142.09234821590687, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273
047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.177
50273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89
.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949,
49, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.1775027
3047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949, 89.17
750273047949, 89.17750273047949, 89.17750273047949, 89.17750273047949]
```

In [20]:
```
plt.figure(figsize=(10,4))
index3=range(1,51)
sns.lineplot(x=index3, y=accuracy3)
plt.xlabel('No of Iterations')
plt.ylabel('Clustering Objective - Inertia')
plt.title('No of Clusters = 3')
```

Out[20]:
```
Text(0.5, 1.0, 'No of Clusters = 3')
```

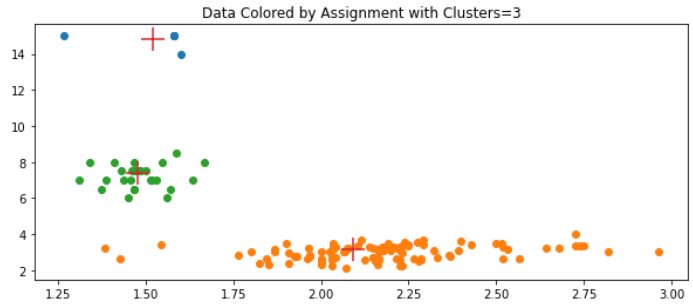


In [21]:
```
print(final_map3)
print("Final Cluster Centers for K=3")
for u in range(len(centroids_x13)):
    print(centroids_x13[u],centroids_x23[u])
```

```
[[5, 6, 15, 16, 17, 19, 21, 22, 23, 26, 31, 40, 41, 43, 44, 45, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 1
06, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 13
5, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149], [9, 12, 32, 34, 37], [0, 1, 2, 3, 4, 7, 8, 10, 11, 13, 14, 18, 20, 24, 2
5, 27, 28, 29, 30, 33, 35, 36, 38, 39, 42, 46, 47, 48, 49]]
Final Cluster Centers for K=3
2.0928950450949397 3.1673129041992576
1.5220456333595596 14.8
1.4788141804347774 7.3678160919540225
```

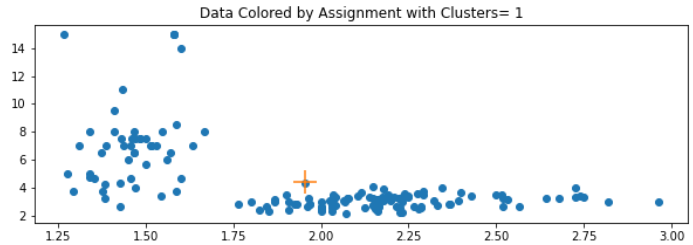
In [22]:
```
x3=[]
y3=[]
for t in range(len(final_map3)):
    pts = fmap[t]
    lx=[]
    ly=[]
    for y in range(len(pts)):
        lx.append(iris_df_new['x1'].iloc[pts[y]])
        ly.append(iris_df_new['x2'].iloc[pts[y]])
    x3.append(lx)
    y3.append(ly)
```

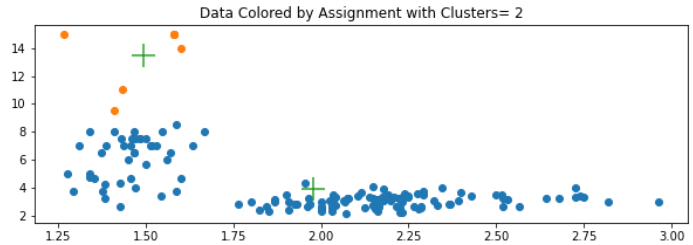
In [23]:
```
# centroids are in + sign

plt.figure(figsize=(10,4))
for u in range(len(x3)):
    plt.scatter(x3[u],y3[u])
plt.scatter(centroids_x13,centroids_x23,marker='+',s=400)
plt.title('Data Colored by Assignment with Clusters=3')
plt.show()
```

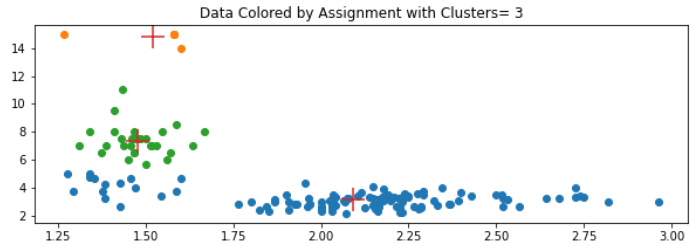


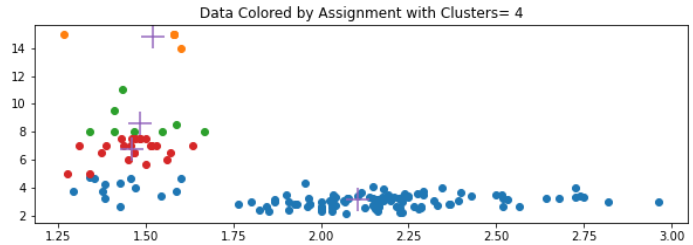
In [24]:
```
X=[]
Y=[]
for u in range(len(maps_of_points)):
    fmap=maps_of_points[u]
    centroids = final_centroids[u]
    mx=[]
    my=[]
    for t in range(len(fmap)):
        pts = fmap[t]
        mn_dist=0
        lx=[]
        ly=[]
        for y in range(len(pts)):
            lx.append(iris_df_new['x1'].iloc[pts[y]])
            ly.append(iris_df_new['x2'].iloc[pts[y]])
        mx.append(lx)
        my.append(ly)
    X.append(mx)
    Y.append(my)
```

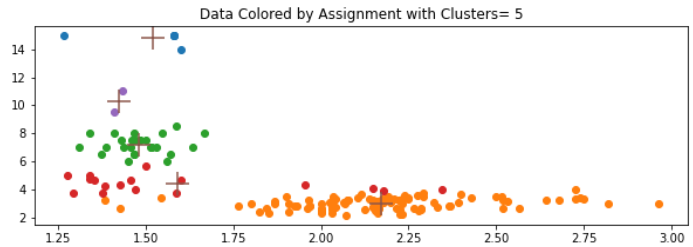
In [25]:
```
for y in range(5):
    plt.figure(figsize=(10,18))
    plt.subplot(5,1,y+1)
    for u in range(len(X[y])):
        plt.scatter(X[y][u],Y[y][u])
    plt.scatter(final_centroids[y][0],final_centroids[y][1],marker='+',s=400)
    plt.title('Data Colored by Assignment with Clusters= '+str(y+1))
    plt.show()
```











In [ ]:

Problem5

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

In [2]:
```
# Training data read
opt_train = pd.read_csv('optdigits.tra',header=None)
```

In [3]:
```
opt_train.head() # showing the training data head
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	64
0	0	1	6	15	12	1	0	0	0	7	...	0	0	0	6	14	7	1	0	0	0
1	0	0	10	16	6	0	0	0	0	7	...	0	0	0	10	16	15	3	0	0	0
2	0	0	8	15	16	13	0	0	0	1	...	0	0	0	9	14	0	0	0	0	7
3	0	0	0	3	11	16	0	0	0	0	...	0	0	0	0	1	15	2	0	0	4
4	0	0	5	14	4	0	0	0	0	0	...	0	0	0	4	12	14	7	0	0	6

5 rows × 65 columns

In [4]:
```
# Function to add regularization in cost calculation
def regularize(reg_param, theta, l):
    weight = np.square(theta)
    weight = np.sum(theta)
    weight = reg_param*weight
    return weight/(2*l)
```

In [5]:
```
# hypothesis function - returns prediction
def hypothesis(theta, X):
    z = np.dot(X, theta)
    return 1/(1+np.exp(-(z))) # sigmoid calculation
```

In [6]:
```
# To calculate the error - cost function
def cost(X, y, theta, reg_param):
    y1 = hypothesis(theta, X)
    return ((-(1/len(X)) * np.sum(y*np.log(y1) + (1-y)*np.log(1-y1)))+regularize(reg_param,theta, len(X)))
```

In [7]:
```
X= pd.concat([pd.Series(1, index=opt_train.index, name='00'), opt_train.iloc[:,:64]], axis=1)
```

In [8]:
```
y = opt_train.iloc[:,64]
```

In [9]:
```
y.unique() # all the classes present
```

Out[9]:
```
array([0, 7, 4, 6, 2, 5, 8, 1, 9, 3])
```

In [10]:
```
y1 = np.zeros([opt_train.shape[0], len(y.unique())])
y1 = pd.DataFrame(y1)
```

In [11]:
```
for i in range(0, len(y.unique())):
    for j in range(0, len(y1)):
        if y[j] == y.unique()[i]:
            y1.iloc[j, i] = 1
        else:
            y1.iloc[j, i] = 0
y1.head() # 1 hot encoded
```

Out[11]:

	0	1	2	3	4	5	6	7	8	9
0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0

In [12]:
```
# function for gradient descent - alpha (learning rate),
# reg_param (regularization parameter), epochs (no of iterations)
# returns optimized theta
def gradient_descent(X, y, theta, alpha, epochs,reg_param):
    m = len(X)
    X = X.to_numpy(dtype='float128')
    X = pd.DataFrame(X)
    for i in range(0, epochs):
        for j in range(0, 10):
            theta = pd.DataFrame(theta)
            # calculating predictions
            h = hypothesis(theta.iloc[:,j], X)
            for k in range(0, theta.shape[0]):
                #updating theta
                theta.iloc[k, j] -= alpha*((reg_param/m)*theta.iloc[k, j]+((1/m)* np.sum((h-y.iloc[:, j])*X.iloc[:, k])))
            theta = pd.DataFrame(theta)
    return theta
```

In [13]:
```
# 1. calculating weights (theta) without regularization parameter (reg_param=0)
# no of iterations chosen = 100
# learning rate 0.02
theta = np.zeros([X.shape[1], y1.shape[1]], dtype='float128')
theta = gradient_descent(X, y1, theta, 0.02, 100,reg_param=0)
```

In [14]:
```
theta.shape
```

Out[14]: (65, 10)

In [15]:
```
X.shape
```

Out[15]: (3823, 65)

In [16]:
```
# predicting output and computing training error
output = []
cost_list=[]
for i in range(0, 10):
    theta1 = pd.DataFrame(theta)
    h = hypothesis(theta1.iloc[:,i], X)
    cst = cost(X,y1.iloc[:, i],theta1.iloc[:,i],reg_param=0)
    output.append(h)
    cost_list.append(abs(cst))
output=pd.DataFrame(output)
```

In [17]:
```
output
```

Out[17]:

	0	1	2	3	4	5	6	7	8	9	...	3813	3814	3815	3816	3817	3818	3819	3820	
0	9.801421e-01	9.974570e-01	0.000194	0.000415	0.048823	0.000108	0.050179	0.003269	0.863145	0.000317	...	0.273924	0.003959	0.012808	0.000271	0.024784	0.022697	2.898723e-02	8.041134e-04	0.0
1	3.714335e-05	9.543595e-06	0.996702	0.002934	0.000005	0.000015	0.000561	0.000115	0.000222	0.000012	...	0.001211	0.003354	0.000008	0.000383	0.000053	0.001036	1.313646e-03	8.499708e-06	0.0
2	9.635947e-03	3.235366e-03	0.005762	0.313275	0.012023	0.000044	0.032320	0.001702	0.015958	0.000339	...	0.996544	0.007664	0.000005	0.000013	0.000095	0.000293	9.991171e-01	9.956169e-02	0.0
3	1.278006e-03	2.384522e-04	0.000047	0.000002	0.909664	0.000016	0.000004	0.000026	0.000070	0.021734	...	0.041906	0.000063	0.000109	0.000004	0.000073	0.000691	7.974308e-02	9.714537e-01	0.9
4	1.105951e-03	3.932455e-03	0.001372	0.000006	0.000422	0.678311	0.000162	0.000448	0.000124	0.000093	...	0.001852	0.010088	0.013013	0.008128	0.012129	0.001368	1.068181e-04	1.594038e-04	0.0
5	4.203992e-02	9.538182e-04	0.000056	0.026336	0.159534	0.001425	0.235538	0.961848	0.429575	0.005877	...	0.006534	0.000061	0.001885	0.068877	0.055251	0.014944	1.853629e-03	8.283574e-04	0.0
6	4.209877e-02	6.685117e-04	0.001680	0.047686	0.142656	0.000059	0.001056	0.008026	0.000171	0.184603	...	0.000422	0.764712	0.012709	0.000139	0.047659	0.031326	1.336275e-03	6.623605e-02	0.0
7	6.401261e-07	3.316930e-08	0.000271	0.135172	0.000117	0.003861	0.054500	0.026109	0.000001	0.003300	...	0.000310	0.235283	0.000045	0.001515	0.001330	0.000013	9.646413e-03	3.093644e-04	0.0
8	1.981562e-02	1.178970e-02	0.000213	0.731998	0.000750	0.060110	0.913917	0.334099	0.000843	0.000962	...	0.000020	0.011138	0.731094	0.005628	0.975997	0.986535	1.840129e-06	5.972493e-07	0.0
9	2.163745e-04	1.712205e-04	0.000839	0.000018	0.000362	0.000155	0.000106	0.002751	0.000018	0.000769	...	0.000001	0.000001	0.001878	0.976590	0.033493	0.032873	2.991920e-07	7.625160e-05	0.0

10 rows × 3823 columns

In [18]:
```
cost_list
```

Out[18]:
```
[0.014801402033700875234,
 0.017657235701162956372,
 0.030844211679009549334,
 0.019518808221721333372,
 0.027644845435902441556,
 0.034113835364826060007,
 0.090094175812511697763,
 0.062570135209355535293,
 0.082670589113625207095,
 0.039239674145673248 9]
```

In [19]:
```
# calculating the accuracy on training data
accuracy = 0
for col in range(0, 10):
    for row in range(len(y1)):
        if y1.iloc[row, col] == 1 and output.iloc[col, row] >= 0.5:
            accuracy += 1
accuracy = accuracy/len(X)
```

In [20]:
```
print(accuracy) # accuracy of training
```

```
0.9134187810619931
```

In [21]:
```
theta # the optimized theta we got after training
```

Out[21]:

	0	1	2	3	4	5	6	7	8	9
0	-0.004624	-0.002724	-0.001323	-0.003980	-0.003156	-0.003105	-0.012970	-0.011154	-0.006924	-0.004276
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2	-0.005358	0.003643	-0.002654	-0.007899	0.003804	-0.000979	-0.018251	-0.015658	-0.007109	0.008604
3	-0.035885	0.009700	-0.031514	-0.082350	0.032649	0.050607	-0.108668	-0.072629	-0.032330	0.009923
4	-0.017527	-0.013714	-0.112049	-0.047031	0.001788	0.006501	-0.059387	-0.053151	-0.000680	-0.009047
...	...	...	...	...	...	...	...	...	...	...
60	-0.009296	-0.065523	-0.116556	-0.049082	0.004739	0.035498	-0.010249	-0.082478	-0.023962	0.025845
61	-0.024606	-0.171349	-0.064290	0.004656	0.014161	-0.040425	-0.018759	-0.051908	-0.021504	0.013214
62	-0.063098	-0.102550	-0.110573	0.083985	0.088518	-0.060681	-0.088281	0.043554	-0.040312	-0.039784
63	-0.048538	-0.031958	-0.048198	0.020569	0.123429	-0.038474	-0.068390	-0.018806	-0.026929	-0.075165
64	-0.004073	-0.002629	-0.001293	-0.009050	0.013497	-0.011272	-0.005718	0.018615	-0.008965	-0.014510

65 rows × 10 columns

In [22]:
```
# reading test data
df_test = pd.read_csv('optdigits.tes',header=None)
df_test_feat = df_test.iloc[:,:64]
df_test_out = df_test.iloc[:,64]
X_test= pd.concat([pd.Series(1, index=df_test_feat.index, name='00'), df_test_feat], axis=1)
y_test = np.zeros([df_test_out.shape[0], len(df_test_out.unique())])
y_test = pd.DataFrame(y_test)
```

In [23]:
```
for i in range(0, len(df_test_out.unique())):
    for j in range(0, len(y_test)):
        if df_test_out[j] == df_test_out.unique()[i]:
            y_test.iloc[j, i] = 1
        else:
            y_test.iloc[j, i] = 0
y_test.head()
#preparing test data
```

Out[23]:

	0	1	2	3	4	5	6	7	8	9
0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0

In [24]:
```
X_test.shape
```

Out[24]:
```
(1797, 65)
```

In [25]:
```
y_test.shape
```

Out[25]:
```
(1797, 10)
```

In [26]:
```
# predicting output and computing test error
output_test = []
cost_test =[]
for i in range(0, 10):
    theta1 = pd.DataFrame(theta)
    h = hypothesis(theta1.iloc[:,i], X_test)
    cst = cost(X_test,y_test.iloc[:, i],theta1.iloc[:,i],reg_param=0)
    output_test.append(h)
    cost_test.append(abs(cst))
output_test=pd.DataFrame(output_test)
```

In [27]:
```
output_test
```

Out[27]:

	0	1	2	3	4	5	6	7	8	9	...	1787	1788	1789	1790	1791	1792	1793	1794
0	9.924354e-01	6.982024e-07	0.000459	0.000088	0.009096	0.000753	0.001083	0.000035	0.000341	0.010524	...	0.001958	6.119043e-03	0.000043	0.000008	1.524805e-03	0.011322	9.526245e-01	0.000068
1	1.542702e-04	7.425269e-04	0.002730	0.000188	0.001906	0.000003	0.000003	0.990806	0.000086	0.000130	...	0.000005	3.135320e-04	0.000623	0.004404	6.516866e-04	0.000039	1.410438e-05	0.000015
2	2.628176e-03	7.504537e-03	0.000592	0.000030	0.995801	0.000103	0.031177	0.016397	0.000119	0.000840	...	0.009288	9.969255e-01	0.003389	0.003411	9.973793e-01	0.000095	9.182113e-04	0.016474
3	4.293137e-04	3.067248e-04	0.001872	0.001001	0.049447	0.000013	0.890591	0.000055	0.000812	0.000091	...	0.000728	1.805603e-02	0.000153	0.000047	1.611867e-02	0.000010	2.802311e-04	0.001974
4	1.334759e-03	1.167886e-03	0.375483	0.007033	0.000030	0.000151	0.000672	0.002060	0.002379	0.000118	...	0.000036	5.173811e-06	0.008161	0.023315	5.748544e-06	0.000066	3.290963e-04	0.000535
5	1.196153e-02	6.621131e-04	0.000002	0.121416	0.000091	0.014741	0.006314	0.002843	0.000027	0.008672	...	0.845222	1.037400e-04	0.000025	0.000005	8.799744e-05	0.018027	2.225992e-03	0.000005
6	1.118866e-02	1.542032e-02	0.162465	0.002173	0.013355	0.001513	0.055968	0.003529	0.976551	0.004061	...	0.001299	5.663010e-04	0.616013	0.096273	1.484730e-03	0.049795	2.066547e-04	0.959605
7	8.782808e-07	9.982649e-01	0.461243	0.019814	0.046129	0.129840	0.053897	0.005195	0.000971	0.004075	...	0.000312	1.296940e-02	0.252830	0.726510	3.288044e-03	0.000286	2.345074e-07	0.149087
8	9.400780e-02	1.267978e-04	0.000016	0.006461	0.000023	0.328184	0.000002	0.003450	0.001314	0.849600	...	0.046028	2.205143e-06	0.000009	0.000075	5.954294e-06	0.959720	1.606514e-04	0.000041
9	6.461856e-04	6.863944e-04	0.000018	0.978065	0.000111	0.175712	0.000013	0.000531	0.005907	0.004287	...	0.003486	6.344543e-07	0.000687	0.017807	1.737959e-07	0.013714	3.547978e-06	0.000051

10 rows × 1797 columns

In [28]:
```
#calculating accuracy on test data using previously obtained theta
accuracy_test = 0
for col in range(0, 10):
    for row in range(len(y_test)):
        if y_test.iloc[row, col] == 1 and output_test.iloc[col, row] >= 0.5:
            accuracy_test += 1
accuracy_test = accuracy_test/len(X_test)
```

In [29]:
```
accuracy_test # test accuracy
```

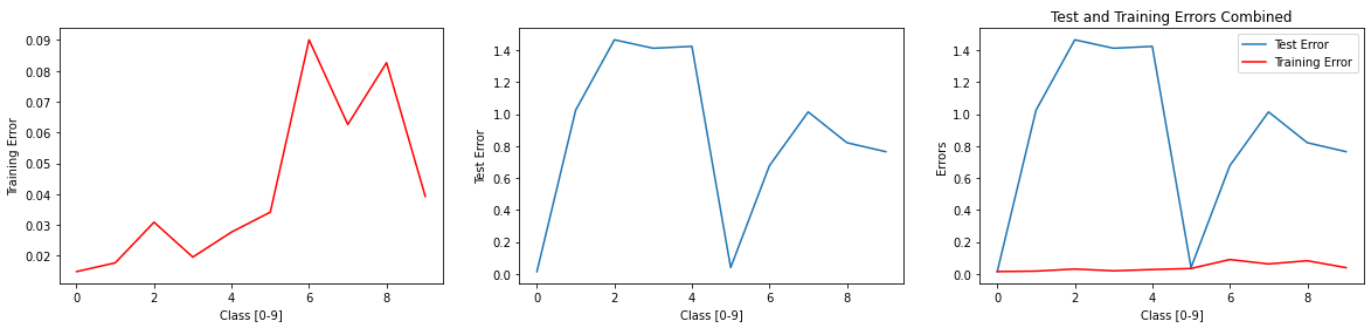
Out[29]:
```
0.19532554257095158
```

In [30]:
```
cost_test
```

Out[30]:
```
[0.014622540741368786402,
 1.0247511057541115017,
 1.4650061284269654717,
 1.4121623293517904182,
 1.4239908186674444586,
 0.040194875684903713614,
 0.6770545836762073491,
 1.0138694249896863227,
 0.82115893094652924817,
 0.76501406048656862813]
```

In [31]:
```
plt.figure(figsize=(20,4))
plt.subplot(1,3,1)
plt.plot(range(0,10),cost_list,'r')
plt.xlabel('Class [0-9]')
plt.ylabel('Training Error')
plt.subplot(1,3,2)
plt.plot(range(0,10),cost_test)
plt.xlabel('Class [0-9]')
plt.ylabel('Test Error')
plt.subplot(1,3,3)
plt.plot(range(0,10),cost_test)
plt.plot(range(0,10),cost_list,'r')
plt.xlabel('Class [0-9]')
plt.legend(["Test Error", "Training Error"], loc ="upper right")
plt.ylabel('Errors')
plt.title('Test and Training Errors Combined')
```

Out[31]:
```
Text(0.5, 1.0, 'Test and Training Errors Combined')
```



In [32]:
```
# 2.) Running the algorithm for 5 different regularization parameters
reg_ind=[10,500,1000,1500,2000] # chosen regularization parameters
theta_list=[] #stores theta values
for j in range(len(reg_ind)):
    theta = np.zeros([X.shape[1], y1.shape[1]], dtype='float128')
    theta = gradient_descent(X, y1, theta, 0.02, 100,reg_param=reg_ind[j])
    theta_list.append(theta)
```

In [33]:
```
# calculate accuracy
def get_accuracy(y,out,x):
    acc = 0
    for col in range(0, 10):
        for row in range(len(y)):
            if y.iloc[row, col] == 1 and out.iloc[col, row] >= 0.5:
                acc += 1
    acc = acc/len(x)
    return acc
```

In [34]:
```
# computing avg training error for models using five different regularization parameters
cost_list_tr=[]
accuracy_list_tr=[]
for j in range(len(reg_ind)):
    ct =0
    out_tr=[]
    theta1 = pd.DataFrame(theta_list[j])
    for i in range(0, 10):
        cst = cost(X,y1.iloc[:, i],theta1.iloc[:,i],reg_ind[j])
        ct+=abs(cst)
        h = hypothesis(theta1.iloc[:,i], X)
        out_tr.append(h)
    out_tr=pd.DataFrame(out_tr)
    accuracy_list_tr.append(get_accuracy(y1,out_tr,X))
    ct=ct/10
    cost_list_tr.append(ct)
print("accuracy of models (training)")
print(accuracy_list_tr)
print("cost ")
print(cost_list_tr)
```

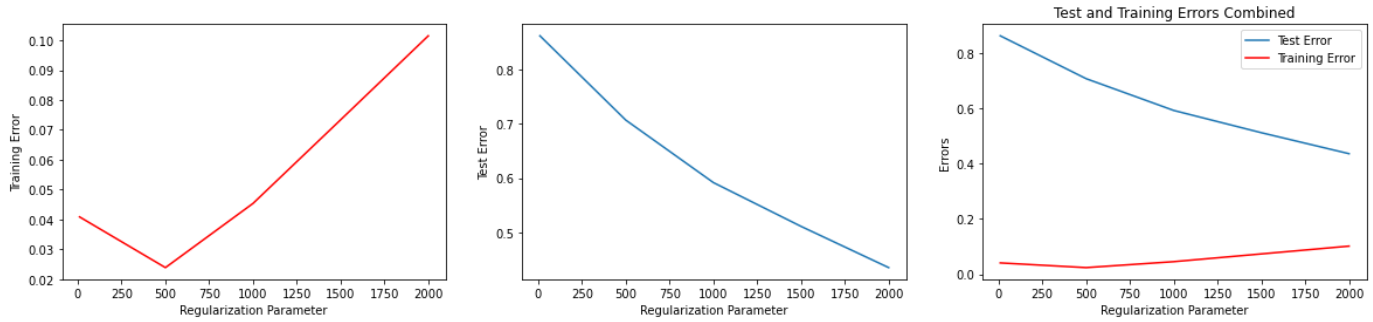
```
accuracy of models (training)
[0.9134187810619931, 0.9139419304211353, 0.8741825791263406, 0.8972011509285901, 0.8830761182317551]
cost
[0.040837146695786431604, 0.023843929858707221865, 0.045355760889616237 61, 0.073373370571553525394, 0.10149471143560085406]
```

In [35]:
```
# computing avg test error for models using five different regularization parameters
cost_list_test=[]
accuracy_list_test=[]
for j in range(len(reg_ind)):
    ct =0
    out_test=[]
    theta1 = pd.DataFrame(theta_list[j])
    for i in range(0, 10):
        cst = abs(cost(X_test,y_test.iloc[:, i],theta1.iloc[:,i],reg_ind[j]))
        h = hypothesis(theta1.iloc[:,i], X_test)
        ct+=cst
        out_test.append(h)
    out_test = pd.DataFrame(out_test)
    accuracy_list_test.append(get_accuracy(y_test,out_test,X_test))
    ct=ct/10
    cost_list_test.append(ct)
print("accuracy of models (training)")
print(accuracy_list_test)
print("cost ")
print(cost_list_test)
```

```
accuracy of models (training)
[0.19532554257095158, 0.19532554257095158, 0.19421257651641624, 0.19532554257095158, 0.19532554257095158]
cost
[0.861324829716324531, 0.7064510370653764231, 0.5916279362821255859, 0.51118692324373028274, 0.435410922454783989]
```

In [36]:
```
plt.figure(figsize=(20,4))
plt.subplot(1,3,1)
plt.plot(reg_ind,cost_list_tr,'r')
plt.xlabel('Regularization Parameter')
plt.ylabel('Training Error')
plt.subplot(1,3,2)
plt.plot(reg_ind,cost_list_test)
plt.xlabel('Regularization Parameter')
plt.ylabel('Test Error')
plt.subplot(1,3,3)
plt.plot(reg_ind,cost_list_test)
plt.plot(reg_ind,cost_list_tr,'r')
plt.xlabel('Regularization Parameter')
plt.legend(["Test Error", "Training Error"], loc ="upper right")
plt.ylabel('Errors')
plt.title('Test and Training Errors Combined')
```

Out[36]:
```
Text(0.5, 1.0, 'Test and Training Errors Combined')
```



In [ ]: