Assignment #: 1

Venkata Sai Chelagamsetty

September 14th, 2022

1. (a) Naive Bayes assumes independence between features i.e the presence of a particular feature in a class is unrelated to the presence of any other feature.Naive Bayes is suitable for solving multi-class prediction problems. If its assumption of the independence of features holds true, it can perform better than other models and requires much less training data.

   (b) KNN is better than logistic regression when boundary is non linear during classifying. KNN supports non-linear solutions where logistic regression supports only linear solutions.

   (c) The entropy for the set is as follows:

$$E(x) = \frac{1}{4}\log 4 + \frac{3}{4}\log\frac{4}{3}$$

   (d) The mean and the standard deviation for the classes is as follows:
   Class A: Mean = 3.4, Standard Deviation = 1.094
   Class B: Mean = 23.4, Standard Deviation = 0.8433
   Note that the standard deviation is calculated using n-1 in the denominator.
   And the priors are as follows:
   Class A: Prior = $\frac{20}{30}$ Class B: Prior = $\frac{10}{30}$

   (e) When $y = 0$:

Variable	**Values**	**Mean**	**Std. Dev**
$x^{(1)}$	4, -4	0	5.656
$x^{(2)}$	7, 5	6	1.414

   When $y = 1$:

Variable	**Values**	**Mean**	**Std. Dev**
$x^{(1)}$	2, 10	6	5.656
$x^{(2)}$	10, 4	7	4.242

   Say that the vector x can take values from the following set (x_1, x_2). Then,

$$P(x = x_1|y = 0) = \frac{1}{\sqrt{2\pi * 5.656}} e^{-\frac{(a_1-0)^2}{2(5.656)^2}}$$

$$P(x = x_2|y = 0) = \frac{1}{\sqrt{2\pi * 1.414}} e^{-\frac{(a_2-6)^2}{2(1.414)^2}}$$

$$P(x = x_1|y = 1) = \frac{1}{\sqrt{2\pi * 5.656}} e^{-\frac{(a_1-6)^2}{2(5.656)^2}}$$

$$P(x = x_2|y = 1) = \frac{1}{\sqrt{2\pi * 4.242}} e^{-\frac{(a_2-7)^2}{2(4.242)^2}}$$

Therefore,

$$P(y=0|x) = \frac{1}{\sqrt{2\pi} * 5.656} e^{-\frac{(x_1-0)^2}{2(5.656)^2}} . \frac{1}{\sqrt{2\pi} * 1.414} e^{-\frac{(x_2-6)^2}{2(1.414)^2}}$$

$$P(y=1|x) = \frac{1}{\sqrt{2\pi} * 5.656} e^{-\frac{(x_1-6)^2}{2(5.656)^2}} . \frac{1}{\sqrt{2\pi} * 4.242} e^{-\frac{(x_2-7)^2}{2(4.242)^2}}$$

2. (a) Let us see which one gives us the highest information gain in the first step. So there are three attributes. Each on of them has only 2 values. So the root of the tree will be split in two based upon the feature values. Let's take the attribute color to be the one that algorithm chooses. Then we will have the following split:

Entropy without any information:

$$E = \frac{7}{16} \log \frac{16}{7} + \frac{9}{16} \log \frac{16}{9} = 0.359$$

Now the entropy in the right leaf(Green):

$$E_r = \frac{2}{3} \log \frac{3}{2} + \frac{1}{3} \log \frac{3}{2} = 0.176$$

Now in the left(Yellow):

$$E_l = \frac{5}{13} \log \frac{13}{5} + \frac{8}{13} \log \frac{13}{8} = 0.289$$

Average of both:

$$E_t = \frac{3}{16} E_r + \frac{13}{16} E_l = 0.268$$

Now, Let's take the attribute size to be the one that algorithm chooses. Then we will have the following split:

Entropy without any information:

$$E = \frac{7}{16} \log \frac{16}{7} + \frac{9}{16} \log \frac{16}{9} = 0.359$$

Now the entropy in the right leaf(Small):

$$E_r = \frac{6}{8} \log \frac{8}{6} + \frac{2}{8} \log \frac{8}{2} = 0.244$$

Now in the left(Large):

$$E_l = \frac{3}{8} \log \frac{8}{3} + \frac{5}{8} \log \frac{8}{5} = 0.287$$

Average of both:

$$E_t = \frac{1}{2} E_r + \frac{1}{2} E_l = 0.265$$

Now, Let's take the attribute shaper to be the one that algorithm chooses. Then we will have the following split:

Entropy without any information:

$$E = \frac{7}{16} \log \frac{16}{7} + \frac{9}{16} \log \frac{16}{9} = 0.359$$

Now the entropy in the right leaf(Irregular):

$$E_r = \frac{1}{4} \log \frac{4}{1} + \frac{3}{4} \log \frac{4}{3} = 0.244$$

Now in the left(Regular):

$$E_l = \frac{6}{12} \log \frac{12}{6} + \frac{6}{12} \log \frac{12}{6} = 0.301$$

Average of both:

$$E_t = \frac{4}{16}E_r + \frac{12}{16}E_l = 0.286$$

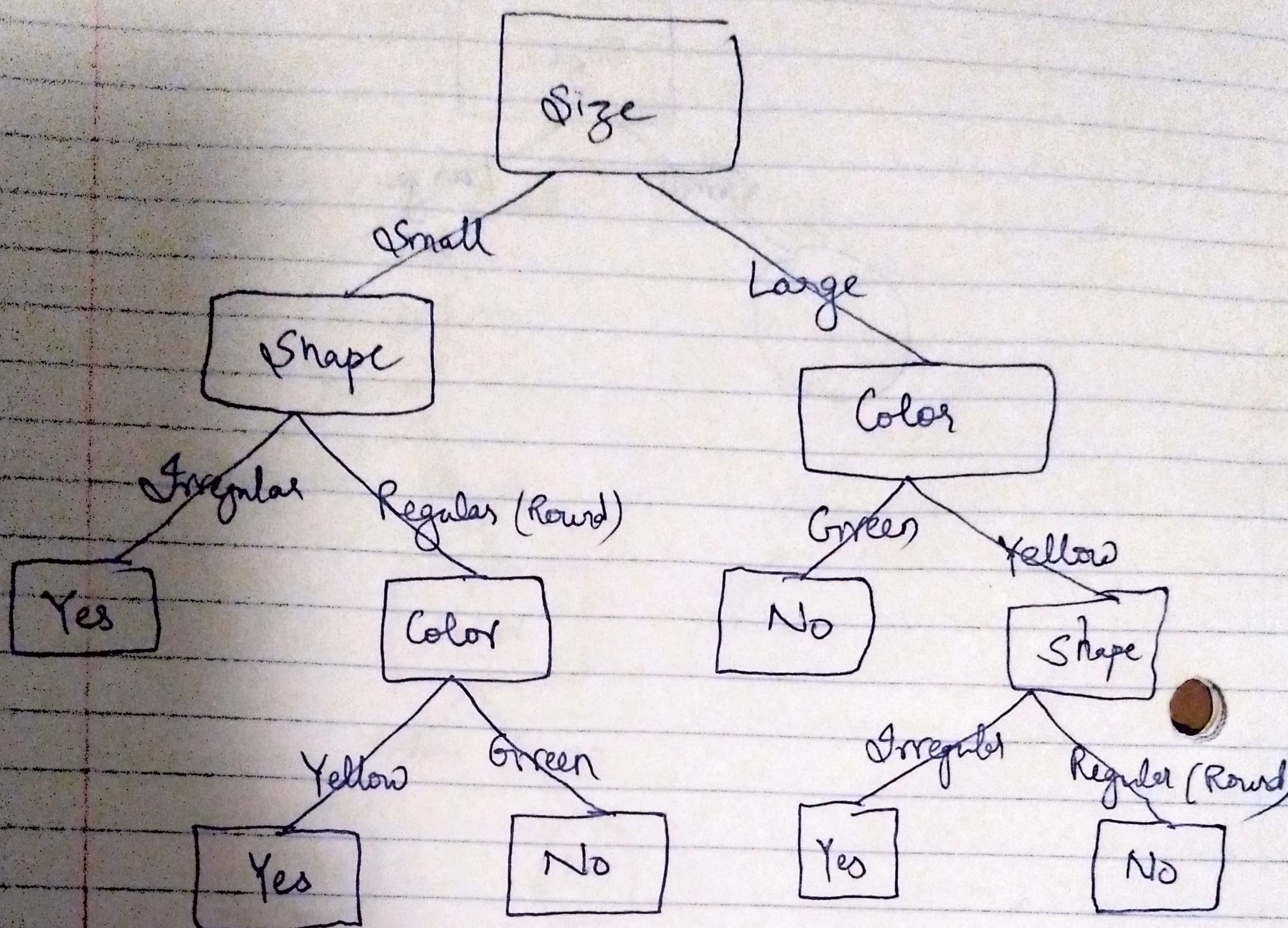
Therefore we will be choosing the size attribute. (As the inital entropy is same for all, we will choose the function with minimum avearge entropy after division)

(b) Having done the above calculations, in a similar way we proceed with other attributes and arrive at the following decision tree.

(c) Yes, some problems do arise. If each value is treated as a discrete value, we will have a very large tree. This results in overfitting of the data. We aren't able to set any thresholds. If we go to the testing phase, only if the incoming value is out of one of the training samples, we can predict correctly. Without generalization, we aren't able to predict the correct outputs leading to high test error.

(d) Code

(e) No standardization done.


Size
Small
Large
Shape
Color
Irregular
Regular (Round)
Green
Yellow
Yes
Color
No
Shape
Yellow
Green
Irregular
Regular (Round)
Yes
No
Yes
No

In [ ]:
```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.preprocessing import StandardScaler
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mo
unt("/content/drive", force_remount=True).
```

In [ ]:
```
df = pd.read_csv("/content/drive/MyDrive/Datasets/Pima.csv",header=None,names=["Pregna
```

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

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunctio
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.00000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.47187
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.33132
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.07800
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.24375
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.37250
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.62625
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.42000

In [ ]:
```
print(df.shape)
```

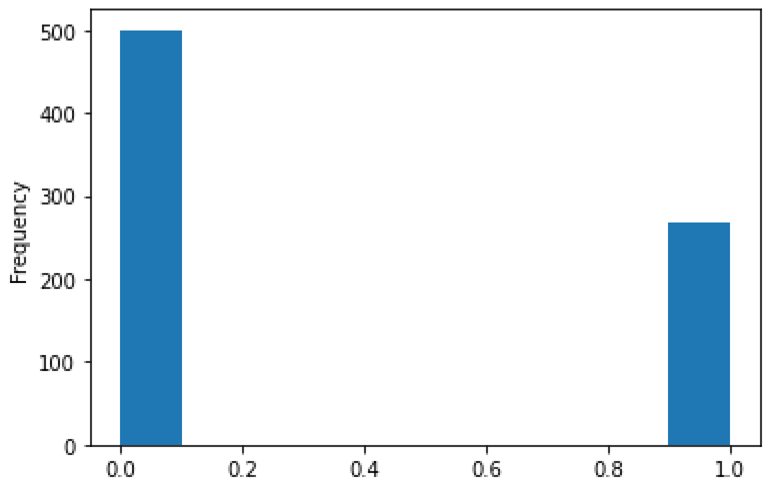
```
(768, 9)
```

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Y'],
      dtype='object')
```

In [ ]:
```
df["Y"].plot.hist()
```

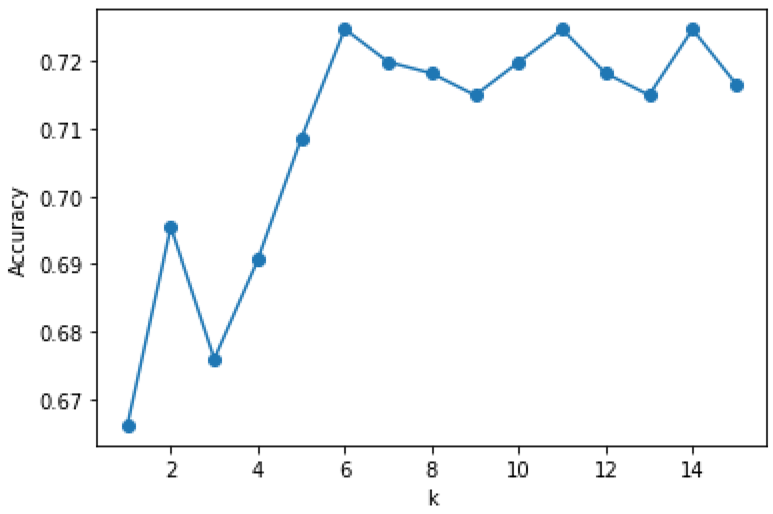
Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f54dc86bad0>
```



In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:,0:8],df["Y"], test_size=
```

In [ ]:
```
accur=[]
x=[]
for k in range(1,16):
  knn = KNeighborsClassifier(n_neighbors=k)
  dic = cross_validate(knn,X_train, y_train, cv=5)
  accur.append(np.mean(dic["test_score"]))
  x.append(k)
plt.plot(x,accur,marker='o')
plt.xlabel("k")
plt.ylabel("Accuracy")
```

Out[ ]:
```
Text(0, 0.5, 'Accuracy')
```



In [ ]:
```
print(accur[6], accur[11], accur[14])
```

```
0.7199386911901906 0.7182593629214982 0.7166600026656005
```

As clearly seen from the graphs and the printed values above, picking k = 6 is the best option. Even though k=11, 14 come close to k=6 in accuracy, we'll be only increasing the running time of the algorithm

In [ ]:
```
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(knn.score(X_test,y_test))
print(f"The test error is {(100*(1-knn.score(X_test,y_test))):.3f}%")
```

```
0.7792207792207793
The test error is 22.078%
```

In [ ]:
```
scaler = StandardScaler()
x_train = scaler.fit_transform(X_train, y_train)
scaler = StandardScaler()
x_test = scaler.fit_transform(X_test, y_test)
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(x_train, y_train)
y_pred = knn.predict(x_test)
print(knn.score(x_test,y_test))
print(f"The test error is {(100*(1-knn.score(x_test,y_test))):.3f}%")
```

```
0.7922077922077922
The test error is 20.779%
```

Yes, centralization and standardization do impact the accuracy. Standardizing the features around the center and 0 with a standard deviation of 1 is important when we compare measurements that have different values. Variables that are measured at different scales do not contribute equally to the analysis and might end up creating a bais.

In [208…

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_validate
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import random
```

In [ ]:

```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mo
unt("/content/drive", force_remount=True).
```

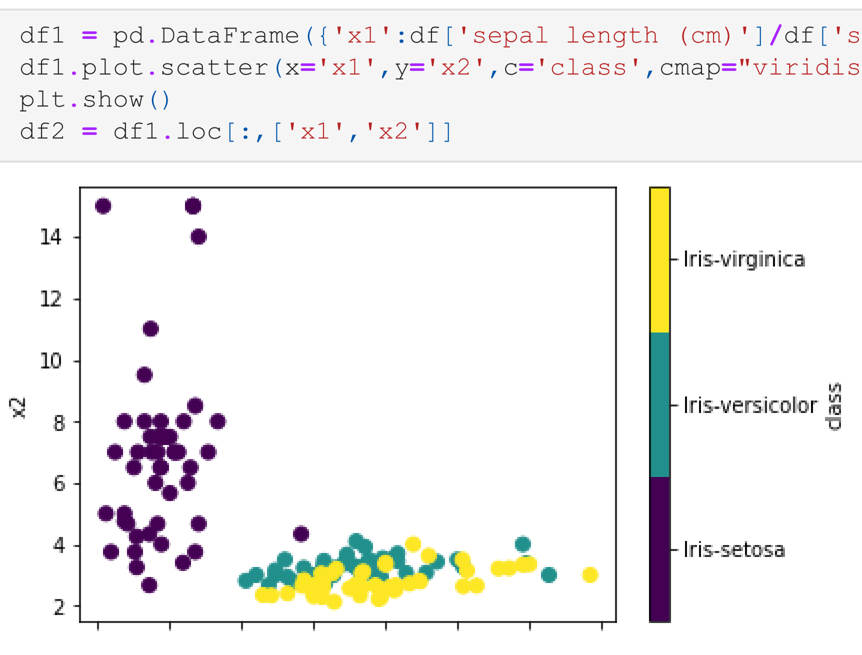
In [ ]:

```python
df = pd.read_csv("/content/drive/MyDrive/Datasets/iris.data", header=None, names=["sep
df.head()
print(df['sepal length (cm)']/df['sepal width (cm)'])
```

```
0      1.457143
1      1.633333
2      1.468750
3      1.483871
4      1.388889
         ...
145    2.233333
146    2.520000
147    2.166667
148    1.823529
149    1.966667
Length: 150, dtype: float64
```

In [ ]:

```python
df1 = pd.DataFrame({'x1':df['sepal length (cm)']/df['sepal width (cm)'],'x2':df['petal
df1.plot.scatter(x='x1',y='x2',c='class',cmap="viridis", s=50)
plt.show()
df2 = df1.loc[:,['x1','x2']]
```



In [ ]:

```python
def dist(p,q):
  d = np.linalg.norm(p - q)
  return d
```

In [ ]:

```python
def kmeansp(nclusters, df, ite):
  j = random.randint(0,len(df))
  c1 = [df.loc[j,'x1'],df.loc[j,'x2']]
  c1 = np.array(c1)
  centers = []
  #print(c1)
  centers.append(c1)
  i=0
  y = np.zeros(len(df))
  #print(df.iloc[0])
  v =[ ]
  v.append(j)
  for k in range(1, nclusters):
    c2 = df.iloc[0].to_numpy()
    c2 = np.array(c2)
    #print(c2)
    min = dist(np.array(np.mean(centers,axis=0)), c2)
    for r in range(1, len(df)):
      #print(r)
      if r in v:
        continue
      if dist(np.array(np.mean(centers,axis=0)), (df.iloc[r]).to_numpy())>min:
        c2 = (df.iloc[r]).to_numpy()
        j = r
        min = dist(np.array(np.mean(centers,axis=0)), (df.iloc[r]).to_numpy())

    v.append(j)
    #print(np.array(np.mean(centers,axis=0)))
    centers.append(c2)

  #print(centers)
  for q in range(0,ite):
    for r in range(len(df)):

      min = dist(centers[0], (df.iloc[r]).to_numpy())
      for k in range(1,nclusters):
        if dist(centers[k], (df.iloc[r]).to_numpy())<min:
          min = dist(centers[k], (df.iloc[r]).to_numpy())
          y[r]=k


    #print(y)
    df['result'] = pd.DataFrame(y)
    for k in range(0,nclusters):
      #print((np.mean(df.loc[df['result']==k])).to_numpy())
      mint = (np.mean(df.loc[df['result']==k], axis=0)).to_numpy()
      centers[k][0] = mint[0]
      centers[k][1] = mint[1]

    df.drop(['result'],axis=1,inplace=True)

  return (y, centers)
```

In [ ]:

```python
#df2.drop('result',axis=1,inplace=True)
print(df2)
y, centers = kmeansp(4, df2, 50)
print(y)
```

```
           x1        x2
0    1.457143  7.000000
1    1.633333  7.000000
2    1.468750  6.500000
3    1.483871  7.500000
4    1.388889  7.000000
..        ...       ...
145  2.233333  2.260870
146  2.520000  2.631579
147  2.166667  2.600000
148  1.823529  2.347826
149  1.966667  2.833333

[150 rows x 2 columns]
[0. 0. 0. 1. 0. 2. 0. 1. 0. 3. 1. 1. 3. 1. 0. 2. 2. 0. 0. 0. 1. 2. 0. 2.
 1. 1. 2. 1. 0. 1. 1. 2. 3. 0. 3. 0. 0. 3. 0. 1. 2. 2. 0. 2. 0. 0. 1. 0.
 1. 0. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2.]
```

In [ ]:

```python
kmean = KMeans(n_clusters=3,init='k-means++',random_state=0)
y = kmean.fit_predict(df2)
print(df2)
print(y)
```

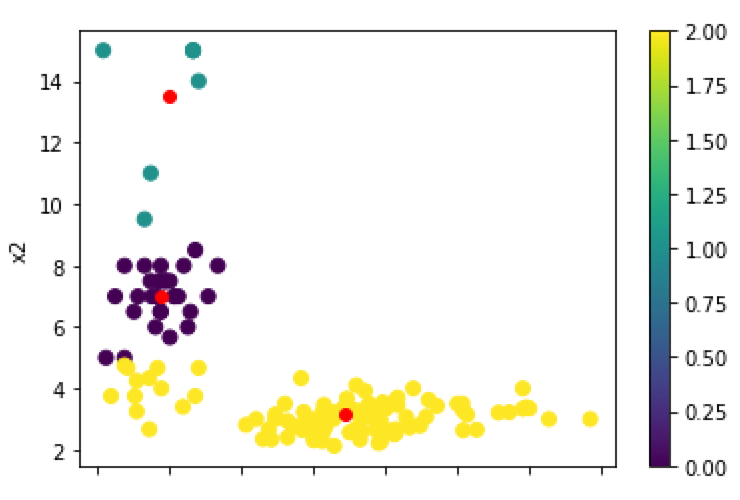
```
           x1        x2
0    1.457143  7.000000
1    1.633333  7.000000
2    1.468750  6.500000
3    1.483871  7.500000
4    1.388889  7.000000
..        ...       ...
145  2.233333  2.260870
146  2.520000  2.631579
147  2.166667  2.600000
148  1.823529  2.347826
149  1.966667  2.833333

[150 rows x 2 columns]
[1 1 1 1 1 2 2 1 1 0 1 1 0 0 1 2 2 2 1 2 1 2 2 2 1 1 2 1 1 1 1 2 0 1 0 1 1
 0 1 1 2 2 1 2 2 2 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

In [ ]:

```python
df2.plot.scatter(x='x1',y='x2',c=y,cmap="viridis", s=50)
plt.plot(centers[0][0],centers[0][1],'ro')
plt.plot(centers[1][0],centers[1][1],'ro')
plt.plot(centers[2][0],centers[2][1],'ro')
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7fd8e5c91890>]
```



In [ ]:

```python
def objec(X, y, k):
  df = X
  obj=[]
  df['result'] = y.tolist()
  for i in range(0,k):
    m1 = np.mean(df.loc[df['result']==i,'x1'])
    m2 = np.mean(df.loc[df['result']==i,'x2'])
    obj.append(((df.loc[df['result']==i,'x1']-m1)**2+(m2-df.loc[df['result']==i,'x2'])
  df.drop('result',axis=1,inplace=True)
  #X.drop('result',axis=1)
  return np.mean(obj)
```

In [ ]:

```python
y, centers = kmeansp(6, df2, 1)
print(y)
obj = objec(df2, np.asarray(y), 3)
print(df2)
print(obj)
objective = []
avg=[]
obj=[]
for i in range(1, 49):

    y, centers = kmeansp(1, df2, i)
    obj.append(objec(df2, np.asarray(y), k))

avg.append(np.mean(obj))


    #print(obj)
#for k in range(1,6):
  #avg=[]
  #for i in range(1,50):

    #y, centres = kmeansp(k, df2, i)
    #avg.append(objec(df2, np.asarray(y), k))
  #objective.append(np.mean(avg))
```

```
[0. 0. 0. 0. 0. 4. 0. 0. 0. 3. 0. 0. 3. 0. 0. 4. 2. 0. 0. 0. 0. 4. 0. 4.
 0. 0. 4. 0. 0. 0. 0. 4. 1. 0. 3. 0. 0. 3. 0. 0. 4. 4. 0. 2. 0. 0. 0. 0.
 0. 0. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 2. 4.
 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 2. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4.
 4. 4. 4. 4. 2. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 2. 2. 4. 4. 4. 4.
 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 4. 2. 4. 4. 4. 4. 4. 4. 4.
 2. 4. 4. 4. 2. 4.]
           x1        x2
0    1.457143  7.000000
1    1.633333  7.000000
2    1.468750  6.500000
3    1.483871  7.500000
4    1.388889  7.000000
..        ...       ...
145  2.233333  2.260870
146  2.520000  2.631579
147  2.166667  2.600000
148  1.823529  2.347826
149  1.966667  2.833333

[150 rows x 2 columns]
22.25353323271688
```

In [ ]:

```python
obj=[]
for i in range(1, 49):

    y, centers = kmeansp(2, df2, i)
    obj.append(objec(df2, np.asarray(y), k))

avg.append(np.mean(obj))
```

In [ ]:

```python
obj=[]
for i in range(1, 49):

    y, centers = kmeansp(3, df2, i)
    obj.append(objec(df2, np.asarray(y), k))

avg.append(np.mean(obj))
```

In [ ]:

```python
obj=[]
for i in range(1, 49):

    y, centers = kmeansp(5, df2, i)
    obj.append(objec(df2, np.asarray(y), k))

avg.append(np.mean(obj))
```

In [ ]:

```python
obj=[]
for i in range(1, 48):

    y, centers = kmeansp(4, df2, i)
    obj.append(objec(df2, np.asarray(y), k))

avg.append(np.mean(obj))
```

In [ ]:

```python
y, centers = kmeansp(4, df2, 49)
obj.append(objec(df2, np.asarray(y), k))
avg[4] = np.mean(obj)
print(avg)
```

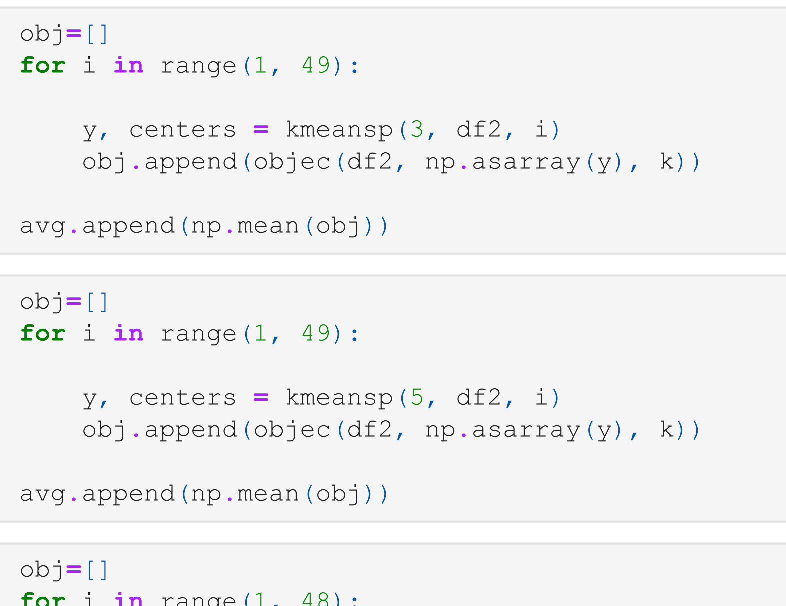
```
[1071.2286861568834, 421.4320610310495, 34.343314954367926, 18.130100864385177, 19.328
216175883714]
```

In [ ]:

```python
plt.plot([x for x in range(1,6)],avg, '-o')
plt.xlabel("k")
plt.ylabel("Clustering Objective")
```

Out[ ]:

```
Text(0, 0.5, 'Clustering Objective')
```



Note that here we have taken the distance measure as the clustering objective. As we need a comparitive study to determine the best k, means work as all we need is comparision. But while plottimg the accuracy vs no. of iterations curve below, we take the exact clustering objective.

According to this, k=3 can be taken following the elbow method. The elbow method involves selecting the value of k that maximizes explained variance while minimizing K; that is, the value of k at the crook of the elbow. The technical sense underlying this is that a minimal gain in explained variance at greater values of k is offset by the increasing risk of overfitting. As k=3, the graph corresponing to that has already been shown above.

In [ ]:

```python
obj=[]
y, centers = kmeansp(3, df2, 50)
print(centers)
y, centers = kmeansp(3, df2, 40)
print(centers)
def objec(X, y, k, centers):
  df = X
  obj=[]
  df['result'] = y.tolist()
  for i in range(0,k):
    m1 = centers[k][0]
    m2 = centers[k][1]
    obj.append(((df.loc[df['result']==i,'x1']-m1)**2+(m2-df.loc[df['result']==i,'x2'])
  df.drop('result',axis=1,inplace=True)
  #X.drop('result',axis=1)
  return np.mean(obj)

obj=[]
for i in range(1, 100):

    y, centers = kmeansp(3, df2, i)
    obj.append(objec(df2, np.asarray(y), k,  centers))
```

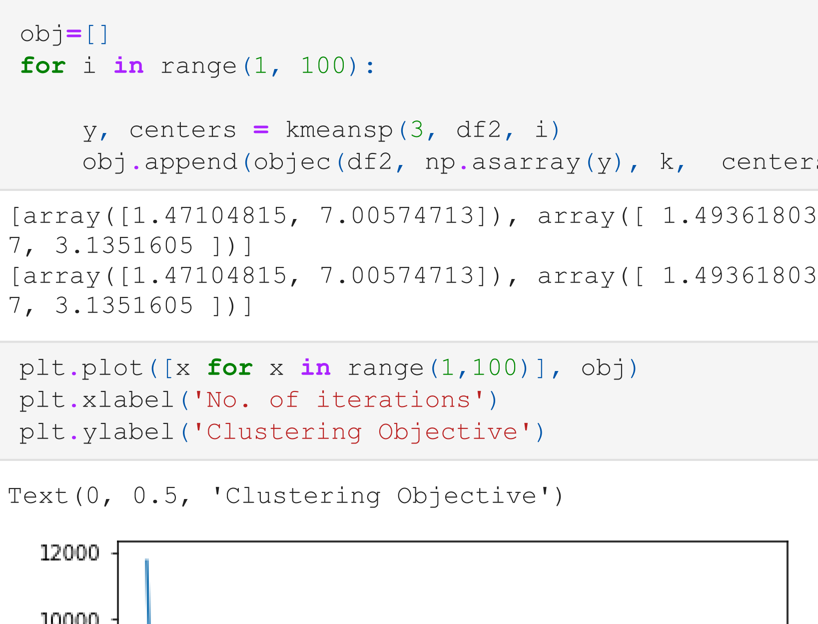
```
[array([1.47104815, 7.00574713]), array([ 1.49361803, 13.5        ]), array([2.1066310
7, 3.1351605 ])]
[array([1.47104815, 7.00574713]), array([ 1.49361803, 13.5        ]), array([2.1066310
7, 3.1351605 ])]
```

In [ ]:

```python
plt.plot([x for x in range(1,100)], obj)
plt.xlabel('No. of iterations')
plt.ylabel('Clustering Objective')
```

Out[ ]:

```
Text(0, 0.5, 'Clustering Objective')
```



As we see that with an increase in the no. of iterations, the clustering objective which we have choosen more or less reaches steady state. The spikes are probably due to the computaion limit of the program.

In [ ]:

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from scipy.special import softmax
onehot_encoder = OneHotEncoder(sparse=False)
```

In [2]:
```
df = pd.read_csv('/content/drive/MyDrive/Datasets/optdigits.tra',header=None)
df.head()
```

Out[2]:

	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 [3]:
```
def loss(X, Y, W):
    """
    Y: onehot encoded
    """
    Z = - X @ W
    N = X.shape[0]
    loss = 1/N * (np.trace(X @ W @ Y.T) + np.sum(np.log(np.sum(np.exp(Z), axis=1))))
    return loss

def gradient(X, Y, W, mu):
    """
    Y: onehot encoded
    """
    Z = - X @ W
    P = softmax(Z, axis=1)
    N = X.shape[0]
    gd = 1/N * (X.T @ (Y - P)) + 2 * mu * W
    return gd

def gradient_descent(X, Y, max_iter=1000, eta=0.1, mu=0.01):
    """
    Very basic gradient descent algorithm with fixed eta and mu
    """
    Y_onehot = onehot_encoder.fit_transform(Y.reshape(-1,1))
    W = np.zeros((X.shape[1], Y_onehot.shape[1]))
    step = 0
    step_lst = []
    loss_lst = []
    acc = []
    W_lst = []

    while step < max_iter:
        step += 1
        W -= eta * gradient(X, Y_onehot, W, mu)
        step_lst.append(step)
        W_lst.append(W)
        loss_lst.append(loss(X, Y_onehot, W))
        acc.append(100*(1-loss(X, Y_onehot, W)))

    df = pd.DataFrame({
        'step': step_lst,
        'loss': loss_lst,
        'acc':acc
    })
    return df, W

class Multiclass:
    def fit(self, X, Y):
        self.loss_steps, self.W = gradient_descent(X, Y)

    def acc_plot(self):
        return self.loss_steps.plot(
            x='step',
            y='acc',
            xlabel='Step',
            ylabel='Accuracy'
        )

    def loss_plot(self):
        return self.loss_steps.plot(
            x='step',
            y='loss',
            xlabel='step',
            ylabel='loss'
        )

    def predict(self, H):
        Z = - H @ self.W
        P = softmax(Z, axis=1)
        return np.argmax(P, axis=1)
```

In [4]:
```
X = (df.iloc[:, 0:64])
x = X.to_numpy()
Y = df.iloc[:, 64]
y = Y.to_numpy()
model = Multiclass()
model.fit(x, y)
```

In [5]:
```
df1 = pd.read_csv("/content/drive/MyDrive/Datasets/optdigits.tes",header=None)
Xt = (df.iloc[:, 0:64])
xt = Xt.to_numpy()
Yt = df.iloc[:, 64]
yt = Yt.to_numpy()
y_pred = model.predict(xt)
```

In [6]:
```
print(model.predict(xt))
print(yt)
```

```
[0 0 7 ... 6 6 7]
[0 0 7 ... 6 6 7]
```

Code inspired from towards data science.

In [8]:
```
```