page-1

May 20, 2024

1 Practical Question 1

Ques1. Create a file "people.txt" with the following data:

Age agegroup height status yearsmarried

21 adult 6.0 single -1

2 child 3 married 0

18 adult 5.7 married 20

221 elderly 5 widowed 2

34 child -7 married 3

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

i) Read the data from the file "people.txt".

[2]:
```
df=pd.read_csv('people.txt')
df
```

[2]:

	Age	agegroup	height	status	yearsmarried
0	21	adult	6.0	single	-1
1	2	child	3.0	married	0
2	18	adult	5.7	married	20
3	221	elderly	5.0	widowed	2
4	34	child	-7.0	married	3

[3]:
```
df1 = np.array(df)
for i in df1:
    print(i)
```

```
[21 'adult' 6.0 'single' -1]
[2 'child' 3.0 'married' 0]
[18 'adult' 5.7 'married' 20]
[221 'elderly' 5.0 'widowed' 2]
[34 'child' -7.0 'married' 3]
```

ii) Create a ruleset E that contain rules to check for the following conditions: 1. The age should be in the range 0-150. 2. The age should be greater than yearsmarried. 3. The status should be married or single or widowed. 4. If age is less than 18 the agegroup should be child, if age is between 18 and 65 the agegroup should be adult, if age is more than 65 the agegroup should be elderly.

[4]:
```
def ruleset_E(data):
    violations = []
    for record in data:
        Age, agegroup, height, status, yearsmarried = record
        if not (0 <= Age <= 150):
            violations.append(f"Age {Age} is not in the range 0-150.")
        if Age <= yearsmarried:
            violations.append(f"Age {Age} is not greater than years married␣
↪({yearsmarried}).")
        if status not in ['single', 'married', 'widowed']:
            violations.append(f"Status {status} is not valid.")
        if Age < 18 and agegroup != 'child':
            violations.append(f"Age {Age} should have agegroup 'child'.")
        elif 18 <= Age <= 65 and agegroup != 'adult':
            violations.append(f"Age {Age} should have agegroup 'adult'.")
        elif Age > 65 and agegroup != 'elderly':
            violations.append(f"Age {Age} should have agegroup 'elderly'.")
    return violations
```

iii) Check whether ruleset E is violated by the data in the file people.txt.

[5]:
```
violations=ruleset_E(df1)
violations
```

[5]:
```
['Age 18 is not greater than years married (20).',
 'Age 221 is not in the range 0-150.',
 "Age 34 should have agegroup 'adult'."]
```

iv) Summarize the results obtained in part (iii)

[6]:
```
if violations:
    print("Ruleset E is violated by the data in people.txt:")
    for violation in violations:
        print(violation)
else:
    print("No violations found in the data.")
```

```
Ruleset E is violated by the data in people.txt:
Age 18 is not greater than years married (20).
Age 221 is not in the range 0-150.
Age 34 should have agegroup 'adult'.
```

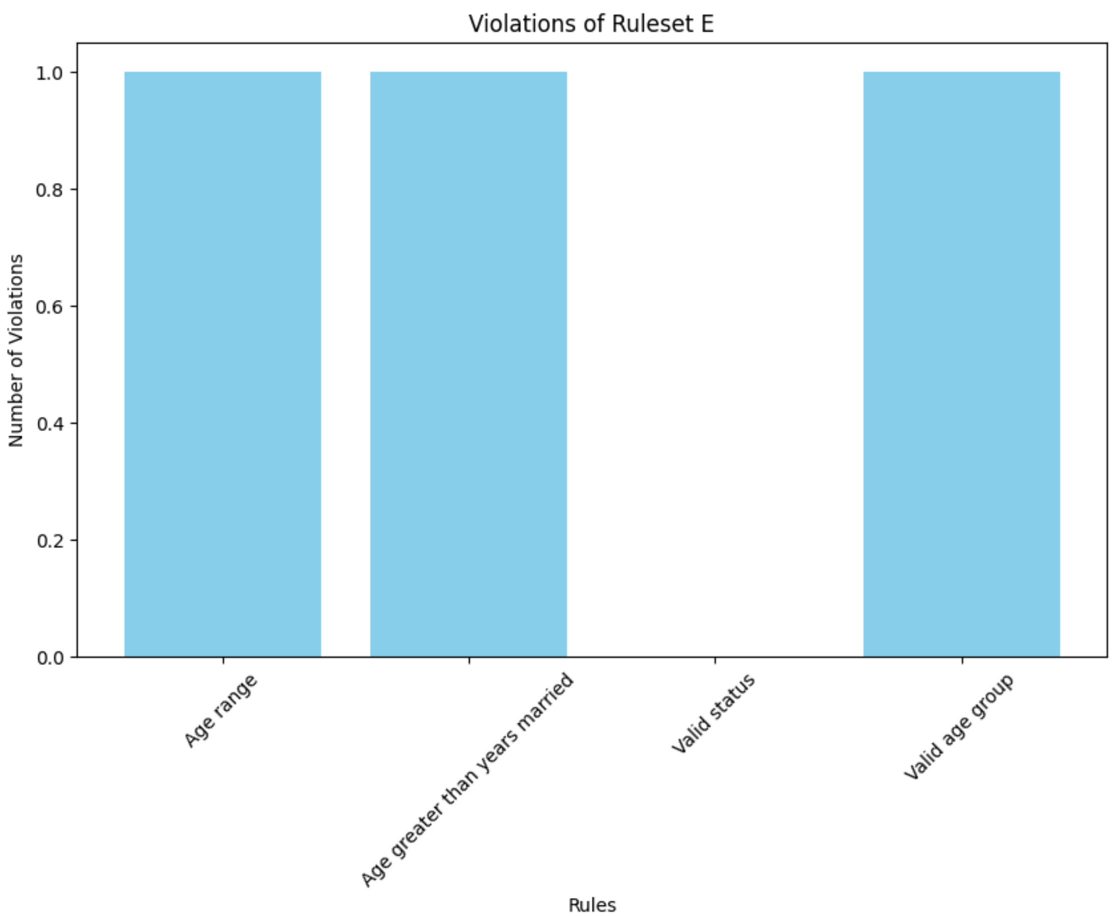
v) Visualize the results obtained in part (iii)

[7]:

```python
import pandas as pd
import matplotlib.pyplot as plt

# Read data from file using read_csv method
def read_data_csv(file_path):
    df = pd.read_csv(file_path, delimiter=' ')
    return df.values.tolist()

# Define ruleset E
def ruleset_E(data):
    violations_count = {'Age range': 0, 'Age greater than years married': 0,␣
↪'Valid status': 0, 'Valid age group': 0}
    for record in data:
        age, agegroup, height, status, yearsmarried = record
        if not (0 <= age <= 150):
            violations_count['Age range'] += 1
        if age <= yearsmarried:
            violations_count['Age greater than years married'] += 1
        if status not in ['single', 'married', 'widowed']:
            violations_count['Valid status'] += 1
        if (age < 18 and agegroup != 'child') or (18 <= age <= 65 and agegroup !
↪= 'adult') or (age > 65 and agegroup != 'elderly'):
            violations_count['Valid age group'] += 1
    return violations_count

# Read data from file
file_path = "people.txt"
data = read_data_csv(file_path)

# Apply ruleset E
violations_count = ruleset_E(df1)

# Summarize results
if sum(violations_count.values()) > 0:
    print("Ruleset E is violated by the data in people.txt:")
    for rule, count in violations_count.items():
        if count > 0:
            print(f"Rule '{rule}': {count} violation(s)")
else:
    print("No violations found in the data.")

# Visualize results
rules = list(violations_count.keys())
counts = list(violations_count.values())

plt.figure(figsize=(10, 6))
plt.bar(rules, counts, color='skyblue')
```

```
plt.xlabel('Rules')
plt.ylabel('Number of Violations')
plt.title('Violations of Ruleset E')
plt.xticks(rotation=45)
plt.show()
```

```
Ruleset E is violated by the data in people.txt:
Rule 'Age range': 1 violation(s)
Rule 'Age greater than years married': 1 violation(s)
Rule 'Valid age group': 1 violation(s)
```



[ ]:

```

```

[ ]:

```

```

2 Practical Question 2

Ques2. Perform the following preprocessing tasks on the dirty_iris datasetii.

[8]:
```
import pandas as pd
import numpy as np
df2 = pd.DataFrame(pd.read_csv("dirty_iris.csv"))
df2
```

[8]:
```
     Unnamed: 0  Sepal.Length  Sepal.Width  Petal.Length  Petal.Width  \
0             1           5.1          3.5           1.4          0.2
1             2           4.9          3.0           1.4          0.2
2             3           4.7          3.2           1.3          0.2
3             4           4.6          3.1           1.5          0.2
4             5           NaN          3.6           1.4          0.2
..          …           …          …           …          …
145         146           6.7          3.0           5.2          2.3
146         147           6.3          2.5           5.0          1.9
147         148           6.5          3.0           5.2          2.0
148         149           6.2          3.4           5.4          2.3
149         150           NaN          3.0           5.1          1.8

       Species
0       Setosa
1       setosa
2       setosa
3       setosa
4       setosa
..         …
145  virginica
146  virginica
147  virginica
148  virginica
149  virginica

[150 rows x 6 columns]
```

i) Calculate the number and percentage of observations that are complete.

[9]:
```
df1 = df2.dropna()
print("No of rows whch are free of null values : ", df1.shape[0])
print("Percentage of rows whch are free of null values : ", (df1.shape[0]/df2.
↪shape[0])*100,"%")
```

```
No of rows whch are free of null values :  131
Percentage of rows whch are free of null values :  87.33333333333333 %
```

ii) Replace all the special values in data with NA.

[10]:
```
df3=df2.replace(to_replace=np.NaN,value="NA")
df3
```

[10]:
```
     Unnamed: 0 Sepal.Length Sepal.Width  Petal.Length  Petal.Width    Species
0             1          5.1         3.5           1.4          0.2     Setosa
1             2          4.9         3.0           1.4          0.2     setosa
2             3          4.7         3.2           1.3          0.2     setosa
3             4          4.6         3.1           1.5          0.2     setosa
4             5           NA         3.6           1.4          0.2     setosa
..          …            …          …             …          …        …
145         146          6.7         3.0           5.2          2.3  virginica
146         147          6.3         2.5           5.0          1.9  virginica
147         148          6.5         3.0           5.2          2.0  virginica
148         149          6.2         3.4           5.4          2.3  virginica
149         150           NA         3.0           5.1          1.8  virginica

[150 rows x 6 columns]
```

iii) Define these rules in a separate text file and read them.

(Use editfile function in R (package editrules). Use similar function in Python).

Print the resulting constraint object.

– Species should be one of the following values: setosa, versicolor or virginica.

– All measured numerical properties of an iris should be positive.

– The petal length of an iris is at least 2 times its petal width.

– The sepal length of an iris cannot exceed 30 cm.

– The sepals of an iris are longer than its petals.

[11]:
```
def species():
    print("Species:")
    print((df2["Species"]=="versicolor") | (df2["Species"]=="virginica") |␣
 ↪(df2["Species"]=="setosa"))
    x=((df2["Species"]=="versicolor") | (df2["Species"]=="virginica") |␣
 ↪(df2["Species"]=="setosa"))
    print("no.of valid values:",x.sum())
    print("no.of valid values:",len(x)-x.sum())
```

[12]:
```
def positive():
    print("Positive value:")
    print((df2["Sepal.Length"]>0) & (df2["Sepal.Width"]>0) & (df2["Petal.
 ↪Length"]>0) & (df2["Petal.Width"]>0))
    x=((df2["Sepal.Length"]>0) & (df2["Sepal.Width"]>0) & (df2["Petal.
 ↪Length"]>0) & (df2["Petal.Width"]>0))
    print("no.of valid values:",x.sum())
    print("no.of valid values:",len(x)-x.sum())
```

[13]:
```
def petal_len():
    print("Petal length:")
    print(df2["Petal.Length"]>=2*df2["Petal.Width"])
    x=(df2["Petal.Length"]>=2*df2["Petal.Width"])
    print("no.of valid values:",x.sum())
    print("no.of valid values:",len(x)-x.sum())
```

[14]:
```
def sepal_petal():
    print("Sepal petal:")
    print(df2["Sepal.Length"]>df2["Petal.Length"])
    x=(df2["Sepal.Length"]>df2["Petal.Length"])
    print("no.of valid values:",x.sum())
    print("no.of valid values:",len(x)-x.sum())
```

[15]:
```
def sepal_len():
    print("sepal length:")
    print(df2["Sepal.Length"]<30)
    x=(df2["Sepal.Length"]<30)
    print("no.of valid values:",x.sum())
    print("no.of valid values:",len(x)-x.sum())
```

[16]:
```
results={species(),positive(),petal_len(),sepal_petal(),sepal_petal()}
```

```
Species:
0      False
1       True
2       True
3       True
4       True
       …
145     True
146     True
147     True
148     True
149     True
Name: Species, Length: 150, dtype: bool
no.of valid values: 145
no.of valid values: 5
Positive value:
0       True
1       True
2       True
3       True
4      False
       …
145     True
146     True
```

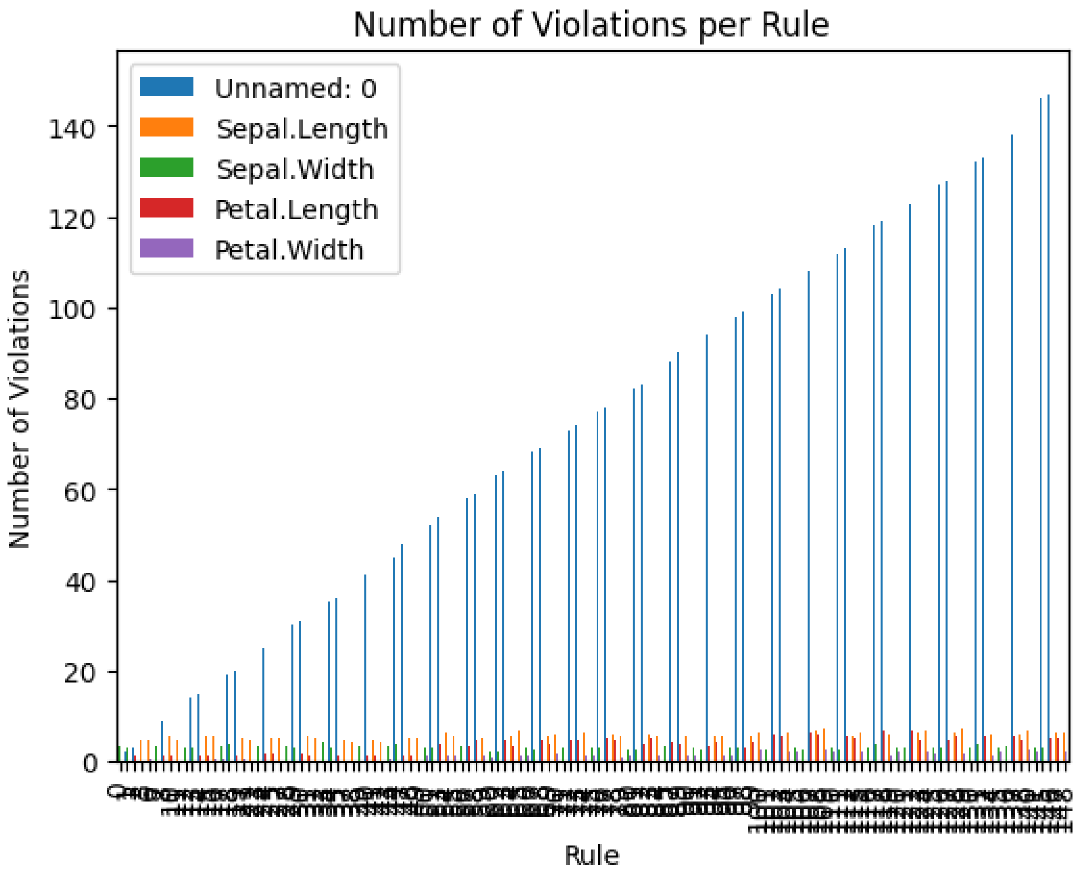
```
147     True
148     True
149    False
Length: 150, dtype: bool
no.of valid values: 131
no.of valid values: 19
Petal length:
0      True
1      True
2      True
3      True
4      True
       …
145    True
146    True
147    True
148    True
149    True
Length: 150, dtype: bool
no.of valid values: 150
no.of valid values: 0
Sepal petal:
0       True
1       True
2       True
3       True
4      False
       …
145     True
146     True
147     True
148     True
149    False
Length: 150, dtype: bool
no.of valid values: 142
no.of valid values: 8
Sepal petal:
0       True
1       True
2       True
3       True
4      False
       …
145     True
146     True
147     True
148     True
149    False
```

```
Length: 150, dtype: bool
no.of valid values: 142
no.of valid values: 8
```

iv) Determine how often each rule is broken (violatedEdits). Also summarize and plot the result.

[17]:
```
# Summarize violations
violations_sum = df1.sum()
print(violations_sum)

# Plot violations
df1.plot(kind='bar')
plt.xlabel('Rule')
plt.ylabel('Number of Violations')
plt.title('Number of Violations per Rule')
plt.show()
```

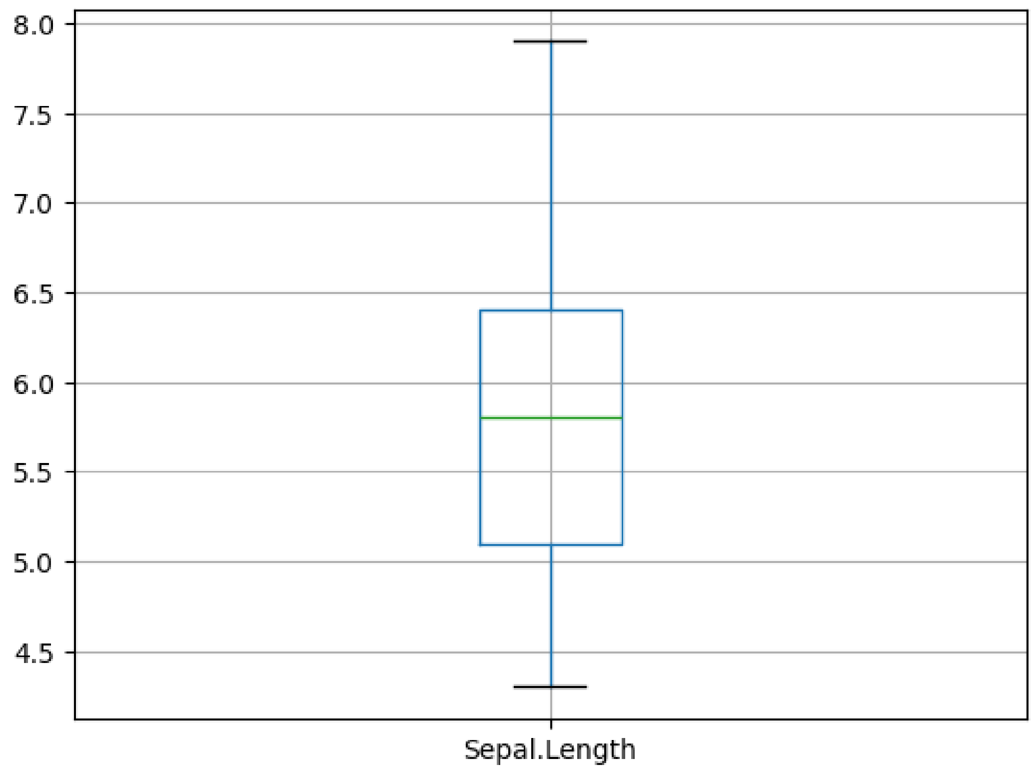
```
Unnamed: 0                                                9864
Sepal.Length                                             766.1
Sepal.Width                                              397.8
Petal.Length                                             496.2
Petal.Width                                              157.6
Species         Setosasetosasetosasetosasetosasetosa SETOSAsetosaseto…
dtype: object
```



v) Find outliers in sepal length using boxplot and boxplot.stats

[18]:
```
df1[["Sepal.Length"]].boxplot()
```

[18]: `<Axes: >`



[ ]:

```

```

[ ]:

```

```

3 Practical Question 3

Ques3: Load the data from wine dataset. Check whether all attributes are standardized or not (mean is 0 and standard deviation is 1). If not, standardize the attributes. Do the same with Iris dataset.

[19]:

```
import numpy as np
import pandas as pd
from sklearn import preprocessing

df=pd.read_csv('wine.csv')
df
```

[19]:

	Wine	Alcohol	Malic.acid	Ash	Acl	Mg	Phenols	Flavanoids	\
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	

```
4        1     13.24         2.59  2.87  21.0  118      2.80         2.69
..    …        …        …    …    …   …        …           …
173      3     13.71         5.65  2.45  20.5   95      1.68         0.61
174      3     13.40         3.91  2.48  23.0  102      1.80         0.75
175      3     13.27         4.28  2.26  20.0  120      1.59         0.69
176      3     13.17         2.59  2.37  20.0  120      1.65         0.68
177      3     14.13         4.10  2.74  24.5   96      2.05         0.76

     Nonflavanoid.phenols  Proanth  Color.int   Hue    OD  Proline
0                    0.28     2.29       5.64  1.04  3.92     1065
1                    0.26     1.28       4.38  1.05  3.40     1050
2                    0.30     2.81       5.68  1.03  3.17     1185
3                    0.24     2.18       7.80  0.86  3.45     1480
4                    0.39     1.82       4.32  1.04  2.93      735
..                    …        …        …    …    …       …
173                  0.52     1.06       7.70  0.64  1.74      740
174                  0.43     1.41       7.30  0.70  1.56      750
175                  0.43     1.35      10.20  0.59  1.56      835
176                  0.53     1.46       9.30  0.60  1.62      840
177                  0.56     1.35       9.20  0.61  1.60      560

[178 rows x 14 columns]
```

[20]:
```
df.std()
```

[20]:
```
Wine                      0.775035
Alcohol                   0.811827
Malic.acid                1.117146
Ash                       0.274344
Acl                       3.339564
Mg                       14.282484
Phenols                   0.625851
Flavanoids                0.998859
Nonflavanoid.phenols      0.124453
Proanth                   0.572359
Color.int                 2.318286
Hue                       0.228572
OD                        0.709990
Proline                 314.907474
dtype: float64
```

[21]:
```
df.mean()
```

[21]:
```
Wine                      1.938202
Alcohol                  13.000618
Malic.acid                2.336348
Ash                       2.366517
```

```
Acl                      19.494944
Mg                       99.741573
Phenols                   2.295112
Flavanoids                2.029270
Nonflavanoid.phenols      0.361854
Proanth                   1.590899
Color.int                 5.058090
Hue                       0.957449
OD                        2.611685
Proline                 746.893258
dtype: float64
```

[22]:
```
df1 = (df-df.mean())/df.std()
df1
```

[22]:
```
         Wine   Alcohol  Malic.acid       Ash       Acl        Mg   Phenols  \
0   -1.210529  1.514341   -0.560668  0.231400 -1.166303  1.908522  0.806722
1   -1.210529  0.245597   -0.498009 -0.825667 -2.483841  0.018094  0.567048
2   -1.210529  0.196325    0.021172  1.106214 -0.267982  0.088110  0.806722
3   -1.210529  1.686791   -0.345835  0.486554 -0.806975  0.928300  2.484437
4   -1.210529  0.294868    0.227053  1.835226  0.450674  1.278379  0.806722
..        …         …           …         …         …         …         …
173  1.370000  0.873810    2.966176  0.304301  0.300954 -0.331985 -0.982841
174  1.370000  0.491955    1.408636  0.413653  1.049555  0.158126 -0.791103
175  1.370000  0.331822    1.739837 -0.388260  0.151234  1.418411 -1.126646
176  1.370000  0.208643    0.227053  0.012696  0.151234  1.418411 -1.030776
177  1.370000  1.391162    1.578712  1.361368  1.498716 -0.261969 -0.391646

     Flavanoids  Nonflavanoid.phenols   Proanth  Color.int       Hue  \
0      1.031908             -0.657708  1.221438   0.251009  0.361158
1      0.731565             -0.818411 -0.543189  -0.292496  0.404908
2      1.212114             -0.497005  2.129959   0.268263  0.317409
3      1.462399             -0.979113  1.029251   1.182732 -0.426341
4      0.661485              0.226158  0.400275  -0.318377  0.361158
..          …                     …         …          …         …
173   -1.420891              1.270726 -0.927563   1.139596 -1.388840
174   -1.280731              0.547563 -0.316058   0.967055 -1.126341
175   -1.340800              0.547563 -0.420888   2.217979 -1.607590
176   -1.350811              1.351077 -0.228701   1.829761 -1.563840
177   -1.270720              1.592131 -0.420888   1.786626 -1.520090

           OD   Proline
0    1.842721  1.010159
1    1.110317  0.962526
2    0.786369  1.391224
3    1.180741  2.328007
4    0.448336 -0.037767
```

```
..         …         …
173 -1.227742 -0.021890
174 -1.481267  0.009866
175 -1.481267  0.279786
176 -1.396759  0.295664
177 -1.424928 -0.593486

[178 rows x 14 columns]
```

[23]:
```python
df1.mean()
```

[23]:
```
Wine                      0.000000e+00
Alcohol                  -9.181170e-16
Malic.acid                0.000000e+00
Ash                      -8.070947e-16
Acl                      -7.983626e-17
Mg                       -1.995907e-17
Phenols                   3.991813e-17
Flavanoids               -3.592632e-16
Nonflavanoid.phenols      3.592632e-16
Proanth                  -1.596725e-16
Color.int                 1.995907e-17
Hue                       1.995907e-16
OD                        3.193450e-16
Proline                  -7.983626e-17
dtype: float64
```

[24]:
```python
from sklearn.datasets import load_iris

iris = load_iris()

X = iris.data
Y = iris.target

iris_df = pd.DataFrame(X, columns=iris.feature_names)
iris_df
```

[24]:
```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 …                …                  …                 …
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
```

```
148                6.2               3.4               5.4               2.3
149                5.9               3.0               5.1               1.8

[150 rows x 4 columns]
```

[25]:

```
iris_df.std()
```

[25]:

```
sepal length (cm)    0.828066
sepal width (cm)     0.435866
petal length (cm)    1.765298
petal width (cm)     0.762238
dtype: float64
```

[26]:

```
iris_df.mean()
```

[26]:

```
sepal length (cm)    5.843333
sepal width (cm)     3.057333
petal length (cm)    3.758000
petal width (cm)     1.199333
dtype: float64
```

[27]:

```
df2 = (iris_df-iris_df.mean())/iris_df.std()
df2

df2.mean()
```

[27]:

```
sepal length (cm)   -1.415442e-15
sepal width (cm)    -1.652752e-15
petal length (cm)   -1.442550e-15
petal width (cm)    -5.543714e-16
dtype: float64
```

[ ]:

```

```

[ ]:

```

```

Section 2: Data Mining Techniques

Run following algorithms on 2 real datasets and use appropriate evaluation measures to compute correctness of obtained patterns:

4 Practical Question 4

Ques4: Run Apriori algorithm to find frequent itemsets and association rules

1.1 Use minimum support as 50% and minimum confidence as 75%

1.2 Use minimum support as 60% and minimum confidence as 60 %

[28]:
```python
market_basket = pd.read_csv('Market_Basket_Optimisation.csv')
market_basket
```

[28]:
```
                shrimp            almonds      avocado    vegetables mix  \
0              burgers          meatballs         eggs               NaN
1              chutney                NaN          NaN               NaN
2               turkey            avocado          NaN               NaN
3        mineral water               milk   energy bar  whole wheat rice
4       low fat yogurt                NaN          NaN               NaN
…                  …                  …            …                  …
7495            butter         light mayo  fresh bread               NaN
7496           burgers  frozen vegetables         eggs      french fries
7497           chicken                NaN          NaN               NaN
7498          escalope          green tea          NaN               NaN
7499              eggs    frozen smoothie  yogurt cake    low fat yogurt

     green grapes whole weat flour yams cottage cheese energy drink  \
0             NaN              NaN  NaN            NaN          NaN
1             NaN              NaN  NaN            NaN          NaN
2             NaN              NaN  NaN            NaN          NaN
3       green tea              NaN  NaN            NaN          NaN
4             NaN              NaN  NaN            NaN          NaN
…            …                …   …              …            …
7495          NaN              NaN  NaN            NaN          NaN
7496    magazines        green tea  NaN            NaN          NaN
7497          NaN              NaN  NaN            NaN          NaN
7498          NaN              NaN  NaN            NaN          NaN
7499          NaN              NaN  NaN            NaN          NaN

     tomato juice low fat yogurt green tea honey salad mineral water salmon  \
0             NaN            NaN       NaN   NaN   NaN           NaN    NaN
1             NaN            NaN       NaN   NaN   NaN           NaN    NaN
2             NaN            NaN       NaN   NaN   NaN           NaN    NaN
3             NaN            NaN       NaN   NaN   NaN           NaN    NaN
4             NaN            NaN       NaN   NaN   NaN           NaN    NaN
…            …              …        …    …    …             …     …
7495          NaN            NaN       NaN   NaN   NaN           NaN    NaN
7496          NaN            NaN       NaN   NaN   NaN           NaN    NaN
7497          NaN            NaN       NaN   NaN   NaN           NaN    NaN
7498          NaN            NaN       NaN   NaN   NaN           NaN    NaN
7499          NaN            NaN       NaN   NaN   NaN           NaN    NaN

     antioxydant juice frozen smoothie spinach  olive oil
0                  NaN             NaN     NaN        NaN
1                  NaN             NaN     NaN        NaN
2                  NaN             NaN     NaN        NaN
3                  NaN             NaN     NaN        NaN
```

```
4                   NaN             NaN     NaN         NaN
…                 …             …     …       …
7495                NaN             NaN     NaN         NaN
7496                NaN             NaN     NaN         NaN
7497                NaN             NaN     NaN         NaN
7498                NaN             NaN     NaN         NaN
7499                NaN             NaN     NaN         NaN

[7500 rows x 20 columns]
```

[29]:
```
clean_data =[]

for i in range(len(market_basket)):
    clean_data.append([x for x in list(map(str,market_basket.iloc[i].tolist()))
 ↪if x != 'nan'])
length = len(clean_data)
print(length)
```

```
7500
```

1.1 Use minimum support as 50% and minimum confidence as 75%

[30]:
```
from efficient_apriori import apriori
itemsets, rules = apriori(clean_data, min_support=0.05, min_confidence=0.075,
 ↪verbosity=1)
```

```
Generating itemsets.
 Counting itemsets of length 1.
  Found 120 candidate itemsets of length 1.
  Found 25 large itemsets of length 1.
 Counting itemsets of length 2.
  Found 300 candidate itemsets of length 2.
  Found 3 large itemsets of length 2.
 Counting itemsets of length 3.
  Found 0 candidate itemsets of length 3.
Itemset generation terminated.

Generating rules from itemsets.
 Generating rules of size 2.
Rule generation terminated.
```

[31]:
```
itemsets
```

[31]:
```
{1: {('burgers',): 654,
  ('eggs',): 1348,
  ('turkey',): 469,
  ('mineral water',): 1787,
```

```
  ('milk',): 972,
  ('whole wheat rice',): 439,
  ('green tea',): 990,
  ('low fat yogurt',): 573,
  ('french fries',): 1282,
  ('soup',): 379,
  ('frozen vegetables',): 715,
  ('spaghetti',): 1306,
  ('cookies',): 603,
  ('cooking oil',): 383,
  ('shrimp',): 535,
  ('chocolate',): 1229,
  ('chicken',): 450,
  ('tomatoes',): 513,
  ('pancakes',): 713,
  ('grated cheese',): 393,
  ('ground beef',): 737,
  ('frozen smoothie',): 474,
  ('escalope',): 595,
  ('cake',): 608,
  ('olive oil',): 493},
 2: {('chocolate', 'mineral water'): 395,
  ('eggs', 'mineral water'): 382,
  ('mineral water', 'spaghetti'): 448}}
```

[32]:
```
for item in sorted(rules, key=lambda item: (item.lift,item.conviction),␣
↪reverse=True):
    print(item)
```

```
{spaghetti} -> {mineral water} (conf: 0.343, supp: 0.060, lift: 1.440, conv:
1.159)
{mineral water} -> {spaghetti} (conf: 0.251, supp: 0.060, lift: 1.440, conv:
1.102)
{chocolate} -> {mineral water} (conf: 0.321, supp: 0.053, lift: 1.349, conv:
1.123)
{mineral water} -> {chocolate} (conf: 0.221, supp: 0.053, lift: 1.349, conv:
1.073)
{eggs} -> {mineral water} (conf: 0.283, supp: 0.051, lift: 1.189, conv: 1.063)
{mineral water} -> {eggs} (conf: 0.214, supp: 0.051, lift: 1.189, conv: 1.043)
```

1.2 Use minimum support as 60% and minimum confidence as 60 %

[33]:
```
itemsets2, rules2 = apriori(clean_data, min_support=0.06, min_confidence=0.06,␣
↪verbosity=1)
```

```
Generating itemsets.
 Counting itemsets of length 1.
  Found 120 candidate itemsets of length 1.
```

```
  Found 21 large itemsets of length 1.
 Counting itemsets of length 2.
  Found 210 candidate itemsets of length 2.
Itemset generation terminated.

Generating rules from itemsets.
Rule generation terminated.
```

[34]:
```python
itemsets2
```

[34]:
```
{1: {('burgers',): 654,
  ('eggs',): 1348,
  ('turkey',): 469,
  ('mineral water',): 1787,
  ('milk',): 972,
  ('green tea',): 990,
  ('low fat yogurt',): 573,
  ('french fries',): 1282,
  ('frozen vegetables',): 715,
  ('spaghetti',): 1306,
  ('cookies',): 603,
  ('shrimp',): 535,
  ('chocolate',): 1229,
  ('chicken',): 450,
  ('tomatoes',): 513,
  ('pancakes',): 713,
  ('ground beef',): 737,
  ('frozen smoothie',): 474,
  ('escalope',): 595,
  ('cake',): 608,
  ('olive oil',): 493}}
```

[35]:
```python
for item in sorted(rules, key=lambda item: (item.lift,item.conviction),
 ↪reverse=True):
        print(item)
```

```
{spaghetti} -> {mineral water} (conf: 0.343, supp: 0.060, lift: 1.440, conv:
1.159)
{mineral water} -> {spaghetti} (conf: 0.251, supp: 0.060, lift: 1.440, conv:
1.102)
{chocolate} -> {mineral water} (conf: 0.321, supp: 0.053, lift: 1.349, conv:
1.123)
{mineral water} -> {chocolate} (conf: 0.221, supp: 0.053, lift: 1.349, conv:
1.073)
{eggs} -> {mineral water} (conf: 0.283, supp: 0.051, lift: 1.189, conv: 1.063)
{mineral water} -> {eggs} (conf: 0.214, supp: 0.051, lift: 1.189, conv: 1.043)
```

[ ]:

```

```

[ ]:

```

```

5 Practical Question 5

Ques5: Use Naive bayes, K-nearest, and Decision tree classification algorithms and build classifiers. Divide the data set into training and test set. Compare the accuracy of the different classifiers under the following situation:

[36]:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split ,␣
↪cross_val_score,KFold,StratifiedKFold
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
```

[37]:

```
iris = load_iris()
X = iris.data
y = iris.target

# Spliting the data into Training set = 75% & Test set = 25%
X_train1, X_test1, y_train1, y_test1 = train_test_split(X, y, test_size=0.25,␣
↪random_state=0)

# Splitting the data into Training set = 66.6% (2/3rd of total) & Test set =33.
↪3%
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, test_size=0.33,␣
↪random_state=0)
```

[38]:

```
# initialize the models
gnb = GaussianNB()
knn = KNeighborsClassifier(n_neighbors=5)
dt = DecisionTreeClassifier(max_depth=3)
```

5.1 a) Training set = 75% Test set = 25% b) Training set = 66.6% (2/3rd of total), Test set = 33.3

[39]:

```
gnb.fit(X_train1, y_train1)
y_pred1 = gnb.predict(X_test1)
print("Naive Bayes accuracy for 75% training set: ", accuracy_score(y_test1,␣
↪y_pred1))

knn.fit(X_train1, y_train1)
y_pred2 = knn.predict(X_test1)
```

```
print("KNN accuracy for 75% training set: ", accuracy_score(y_test1, y_pred2))

dt.fit(X_train1, y_train1)
y_pred3 = dt.predict(X_test1)
print("Decision Tree accuracy for 75% training set: ", accuracy_score(y_test1,␣
↪y_pred3))
```

```
Naive Bayes accuracy for 75% training set:  1.0
KNN accuracy for 75% training set:  0.9736842105263158
Decision Tree accuracy for 75% training set:  0.9736842105263158
```

[40]:
```
# 5.1 b) Training set = 66.6% (2/3rd of total), Test set =33.3%
gnb.fit(X_train2, y_train2)
y_pred4 = gnb.predict(X_test2)
print("Naive Bayes accuracy for 66.6% training set: ", accuracy_score(y_test2,␣
↪y_pred4))

knn.fit(X_train2, y_train2)
y_pred5 = knn.predict(X_test2)
print("KNN accuracy for 66.6% training set: ", accuracy_score(y_test2, y_pred5))

dt.fit(X_train2, y_train2)
y_pred6 = dt.predict(X_test2)
print("Decision Tree accuracy for 66.6% training set: ",␣
↪accuracy_score(y_test2, y_pred6))
```

```
Naive Bayes accuracy for 66.6% training set:  0.96
KNN accuracy for 66.6% training set:  0.98
Decision Tree accuracy for 66.6% training set:  0.94
```

5.2 Training set is chosen by i) hold out method ii) Random subsampling iii) Cross-Validation. Compare the accuracy of the classifiers obtained

[41]:
```
# hold out method
X_train3 , X_test3 , y_train3 , y_test3 = train_test_split(X, y, test_size=0.
↪20, random_state=42)

gnb.fit(X_train3, y_train3)
y_pred7 = gnb.predict(X_test3)
print("Naive Bayes accuracy for hold out method: ", accuracy_score(y_test3,␣
↪y_pred7))

knn.fit(X_train3, y_train3)
y_pred8 = knn.predict(X_test3)
print("KNN accuracy for hold out method: ", accuracy_score(y_test3, y_pred8))

dt.fit(X_train3, y_train3)
y_pred9 = dt.predict(X_test3)
```

```
print("Decision Tree accuracy for hold out method: ", accuracy_score(y_test3,␣
 ↪y_pred9))
```

```
Naive Bayes accuracy for hold out method:  1.0
KNN accuracy for hold out method:  1.0
Decision Tree accuracy for hold out method:  1.0
```

[42]:
```
# Random subsampling method
sum_gnb = 0
sum_knn = 0
sum_dt = 0
for i in range(10):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=i)
    gnb.fit(X_train, y_train)
    y_pred = gnb.predict(X_test)
    sum_gnb += accuracy_score(y_test, y_pred)

    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    sum_knn += accuracy_score(y_test, y_pred)

    dt.fit(X_train, y_train)
    y_pred = dt.predict(X_test)
    sum_dt += accuracy_score(y_test, y_pred)

print("Naive Bayes accuracy for random subsampling method: ", sum_gnb/10)
print("KNN accuracy for random subsampling method: ", sum_knn/10)
print("Decision Tree accuracy for random subsampling method: ", sum_dt/10)
```

```
Naive Bayes accuracy for random subsampling method:  0.9473684210526315
KNN accuracy for random subsampling method:  0.9631578947368421
Decision Tree accuracy for random subsampling method:  0.9447368421052630
```

[43]:
```
# cross-validation
gnb_scores = cross_val_score(gnb, X, y, cv=10)
print("Naive Bayes accuracy for cross-validation: ", gnb_scores.mean())

knn_scores = cross_val_score(knn, X, y, cv=10)
print("KNN accuracy for cross-validation: ", knn_scores.mean())

dt_scores = cross_val_score(dt, X, y, cv=10)
print("Decision Tree accuracy for cross-validation: ", dt_scores.mean())
```

```
Naive Bayes accuracy for cross-validation:  0.9533333333333334
KNN accuracy for cross-validation:  0.9666666666666668
Decision Tree accuracy for cross-validation:  0.96
```

5.3 Data is scaled to standard formats:

[44]:
```
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

x_train4 , x_test4 , y_train4 , y_test4 = train_test_split(X_scaled, y,␣
↪test_size=0.20, random_state=42)

gnb.fit(x_train4, y_train4)
y_pred10 = gnb.predict(x_test4)
print("Naive Bayes accuracy for scaled data: ", accuracy_score(y_test4,␣
↪y_pred10))

knn.fit(x_train4, y_train4)
y_pred11 = knn.predict(x_test4)
print("KNN accuracy for scaled data: ", accuracy_score(y_test4, y_pred11))

dt.fit(x_train4, y_train4)
y_pred12 = dt.predict(x_test4)
print("Decision Tree accuracy for scaled data: ", accuracy_score(y_test4,␣
↪y_pred12))
```

```
Naive Bayes accuracy for scaled data:  1.0
KNN accuracy for scaled data:  1.0
Decision Tree accuracy for scaled data:  1.0
```

[ ]:

[ ]:

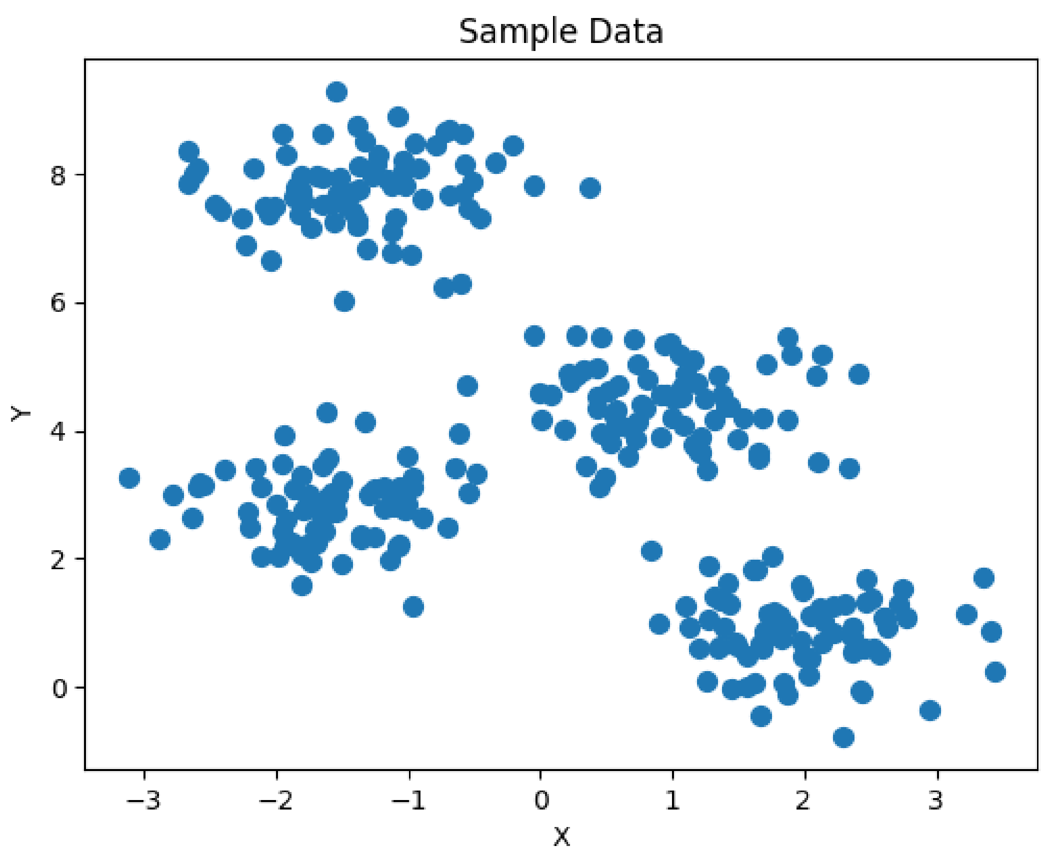
6 Practical Question 6

Ques6: Use Simple Kmeans, DBScan, Hierachical clustering algorithms for clustering. Compare the performance of clusters by changing the parameters involved in the algorithms.

[45]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.metrics import silhouette_score

# Generating sample data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Visualizing the sample data
```

```python
plt.scatter(X[:, 0], X[:, 1], s=50)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Sample Data')
plt.show()

# Simple KMeans Clustering
def kmeans_clustering(X, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters)
    kmeans.fit(X)
    labels = kmeans.labels_
    return labels

# DBSCAN Clustering
def dbscan_clustering(X, eps, min_samples):
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    labels = dbscan.fit_predict(X)
    return labels

# Hierarchical Clustering
def hierarchical_clustering(X, n_clusters):
    agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
    labels = agg_clustering.fit_predict(X)
    return labels

# Evaluating clustering performance using silhouette score
def evaluate_clustering(X, labels):
    if len(np.unique(labels)) > 1:
        silhouette_avg = silhouette_score(X, labels)
        return silhouette_avg
    else:
        return -1  # Return a placeholder value when only noise points are␣
 ↪present

# Testing Simple KMeans with different number of clusters
print("Simple KMeans:")
for n_clusters in range(2, 6):
    labels = kmeans_clustering(X, n_clusters)
    silhouette_avg = evaluate_clustering(X, labels)
    print(f"Number of clusters: {n_clusters}, Silhouette Score:␣
 ↪{silhouette_avg}")

# Testing DBSCAN with different values of epsilon and min_samples
print("\nDBSCAN:")
eps_values = [0.3, 0.5, 0.7, 1.0]
min_samples_values = [5, 10, 15, 20]
for eps in eps_values:
```

```
    for min_samples in min_samples_values:
        labels = dbscan_clustering(X, eps, min_samples)
        silhouette_avg = evaluate_clustering(X, labels)
        if silhouette_avg != -1:
            print(f"EPS: {eps}, Min Samples: {min_samples}, Silhouette Score:␣
↪{silhouette_avg}")
        else:
            print(f"EPS: {eps}, Min Samples: {min_samples}, No valid clusters␣
↪found.")

# Testing Hierarchical Clustering with different number of clusters
print("\nHierarchical Clustering:")
for n_clusters in range(2, 6):
    labels = hierarchical_clustering(X, n_clusters)
    silhouette_avg = evaluate_clustering(X, labels)
    print(f"Number of clusters: {n_clusters}, Silhouette Score:␣
↪{silhouette_avg}")
```




```
Simple KMeans:
Number of clusters: 2, Silhouette Score: 0.5426422297358302
```

```
Number of clusters: 3, Silhouette Score: 0.5890390393551768
Number of clusters: 4, Silhouette Score: 0.6819938690643478
Number of clusters: 5, Silhouette Score: 0.5923875148758644

DBSCAN:
EPS: 0.3, Min Samples: 5, Silhouette Score: -0.02553097772433596
EPS: 0.3, Min Samples: 10, Silhouette Score: -0.25081172884728203
EPS: 0.3, Min Samples: 15, No valid clusters found.
EPS: 0.3, Min Samples: 20, No valid clusters found.
EPS: 0.5, Min Samples: 5, Silhouette Score: 0.6303800996842714
EPS: 0.5, Min Samples: 10, Silhouette Score: 0.5220954071399261
EPS: 0.5, Min Samples: 15, Silhouette Score: 0.3871688267990456
EPS: 0.5, Min Samples: 20, Silhouette Score: 0.2097450211809529
EPS: 0.7, Min Samples: 5, Silhouette Score: 0.559707233404896
EPS: 0.7, Min Samples: 10, Silhouette Score: 0.6569398552813946
EPS: 0.7, Min Samples: 15, Silhouette Score: 0.6198214828463691
EPS: 0.7, Min Samples: 20, Silhouette Score: 0.6045937193339364
EPS: 1.0, Min Samples: 5, Silhouette Score: 0.46285745923867483
EPS: 1.0, Min Samples: 10, Silhouette Score: 0.58977292182092
EPS: 1.0, Min Samples: 15, Silhouette Score: 0.58977292182092
EPS: 1.0, Min Samples: 20, Silhouette Score: 0.58977292182092

Hierarchical Clustering:
Number of clusters: 2, Silhouette Score: 0.54731479631826
Number of clusters: 3, Silhouette Score: 0.58977292182092
Number of clusters: 4, Silhouette Score: 0.6819938690643478
Number of clusters: 5, Silhouette Score: 0.5875473435823221
```

[ ]:

```

```

[ ]:

```

```