nonlinear-dimensionality-reduction

March 26, 2024

1 Nonlinear Dimensionality Reduction

1.0.1 Apply the nonlinear dimensionality reduction methods Locally Linear Embedding (LLE) and ISOMAP to the dataset C, set the number of nearest neighbors to be 5, the projected low dimension to be 4

[259]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import Isomap
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
from sklearn.manifold import LocallyLinearEmbedding
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.preprocessing import LabelEncoder
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

[260]:
```
data = pd.read_csv('DataC.csv')
data.head()
```

[260]:
```
   Unnamed: 0  fea.1  fea.2  fea.3  fea.4  fea.5  fea.6  fea.7  fea.8  fea.9  \
0           1      4      4      3      0      0      4      2      1      4
1           2      5      1      4      3      1      3      5      1      4
2           3      1      3      0      3      1      1      0      1      0
3           4      5      3      2      3      5      2      2      0      4
4           5      3      5      3      3      0      4      1      1      4

   …  fea.776  fea.777  fea.778  fea.779  fea.780  fea.781  fea.782  \
0  …        1        3        0        4        2        1        1
1  …        1        1        3        3        1        3        3
2  …        3        0        2        4        2        2        1
3  …        5        4        5        1        4        4        2
4  …        1        3        3        3        1        2        4

   fea.783  fea.784  gnd
0        4        5    0
1        5        4    0
```

```
2         2         4     0
3         4         4     0
4         1         1     0

[5 rows x 786 columns]
```

[261]:
```python
missing_values = data.isna().sum()
print(missing_values[missing_values > 0])
```

```
Series([], dtype: int64)
```

[262]:
```python
data.describe()
```

[262]:
```
        Unnamed: 0        fea.1        fea.2        fea.3        fea.4  \
count  2066.000000  2066.000000  2066.000000  2066.000000  2066.000000
mean   1033.500000     2.508228     2.547435     2.460794     2.496612
std     596.547148     1.477246     1.502839     1.499851     1.497128
min       1.000000     0.000000     0.000000     0.000000     0.000000
25%     517.250000     1.000000     1.000000     1.000000     1.000000
50%    1033.500000     3.000000     3.000000     2.000000     3.000000
75%    1549.750000     4.000000     4.000000     4.000000     4.000000
max    2066.000000     5.000000     5.000000     5.000000     5.000000

             fea.5        fea.6        fea.7        fea.8        fea.9  …  \
count  2066.000000  2066.000000  2066.000000  2066.000000  2066.000000  …
mean      2.472894     2.490319     2.486447     2.512585     2.522265  …
std       1.509451     1.498071     1.501270     1.524326     1.502456  …
min       0.000000     0.000000     0.000000     0.000000     0.000000  …
25%       1.000000     1.000000     1.000000     1.000000     1.000000  …
50%       2.000000     2.000000     3.000000     3.000000     3.000000  …
75%       4.000000     4.000000     4.000000     4.000000     4.000000  …
max       5.000000     5.000000     5.000000     5.000000     5.000000  …

           fea.776      fea.777      fea.778      fea.779      fea.780  \
count  2066.000000  2066.000000  2066.000000  2066.000000  2066.000000
mean      2.469506     2.522749     2.486447     2.449661     2.498064
std       1.488060     1.515606     1.506422     1.511740     1.496160
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       1.000000     1.000000     1.000000     1.000000     1.000000
50%       2.000000     3.000000     3.000000     2.000000     3.000000
75%       4.000000     4.000000     4.000000     4.000000     4.000000
max       5.000000     5.000000     5.000000     5.000000     5.000000

           fea.781      fea.782      fea.783      fea.784          gnd
count  2066.000000  2066.000000  2066.000000  2066.000000  2066.000000
mean      2.525653     2.542110     2.400290     2.519361     2.035818
std       1.511079     1.491353     1.527783     1.504107     1.398261
```

```
min        0.000000     0.000000     0.000000     0.000000     0.000000
25%        1.000000     1.000000     1.000000     1.000000     1.000000
50%        3.000000     3.000000     2.000000     2.000000     2.000000
75%        4.000000     4.000000     4.000000     4.000000     3.000000
max        5.000000     5.000000     5.000000     5.000000     4.000000

[8 rows x 786 columns]
```

[263]:
```
data_info = data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2066 entries, 0 to 2065
Columns: 786 entries, Unnamed: 0 to gnd
dtypes: int64(786)
memory usage: 12.4 MB
```

[264]:
```
data = data.drop(columns=data.columns[0])
data
```

[264]:
```
      fea.1  fea.2  fea.3  fea.4  fea.5  fea.6  fea.7  fea.8  fea.9  fea.10  \
0         4      4      3      0      0      4      2      1      4       1
1         5      1      4      3      1      3      5      1      4       4
2         1      3      0      3      1      1      0      1      0       2
3         5      3      2      3      5      2      2      0      4       5
4         3      5      3      3      0      4      1      1      4       3
…       …      …      …      …      …      …      …      …      …       …
2061      4      0      3      0      4      0      4      3      1       2
2062      2      2      3      4      2      1      2      3      3       4
2063      2      3      2      3      1      2      5      5      5       0
2064      5      2      4      3      1      0      3      2      2       1
2065      3      3      1      3      2      5      4      2      2       4

      …  fea.776  fea.777  fea.778  fea.779  fea.780  fea.781  fea.782  \
0     …        1        3        0        4        2        1        1
1     …        1        1        3        3        1        3        3
2     …        3        0        2        4        2        2        1
3     …        5        4        5        1        4        4        2
4     …        1        3        3        3        1        2        4
…   …        …        …        …        …        …        …        …
2061  …        0        1        4        5        4        2        2
2062  …        4        0        1        3        4        0        2
2063  …        5        1        1        2        5        2        1
2064  …        3        2        3        1        4        2        4
2065  …        2        3        1        4        4        5        1

      fea.783  fea.784  gnd
0           4        5    0
```

```
1           5        4    0
2           2        4    0
3           4        4    0
4           1        1    0
…       …       …   …
2061        2        2    4
2062        3        2    4
2063        1        3    4
2064        3        4    4
2065        3        1    4

[2066 rows x 785 columns]
```

[265]:
```
x = data.iloc[:,:-1].values
y = data.iloc[:,-1].values
```

[266]:
```
print("Classes present:")
print(set(y))
```

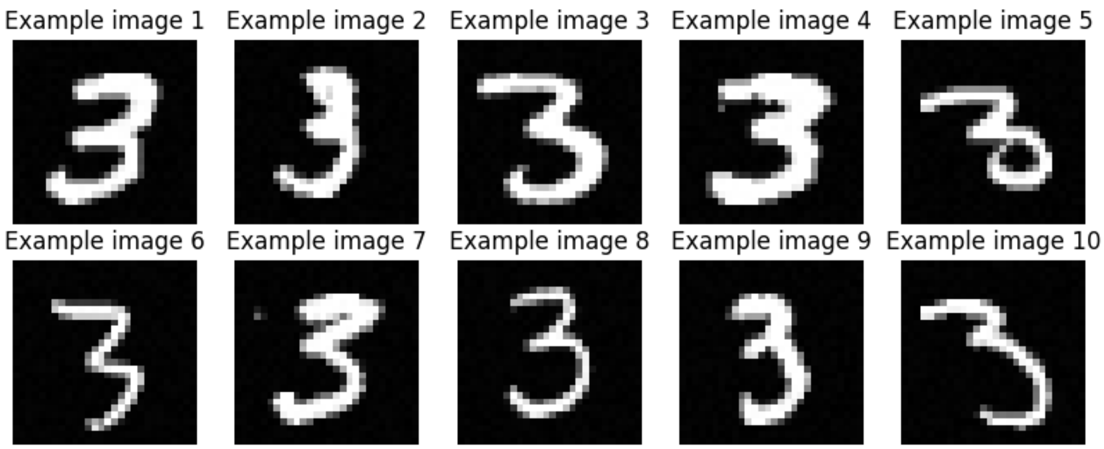
```
Classes present:
{0, 1, 2, 3, 4}
```

[267]:
```
data_3 = data[data['gnd']==3]
x = data_3.iloc[:,:-1].values
y = data_3.iloc[:,-1].values
```

1.0.2 Q1. Apply LLE to the images of digit '3' only. Visualize the original images by plotting the images corresponding to those instances on 2-D representations of the data based on the first and second components of LLE. Describe qualitatively what kind of variations is captured.

[268]:
```
plt.figure(figsize=(10, 10))
for i in range(10):  # Plotting first 5 images
  plt.subplot(5, 5, i + 1)
  plt.imshow(x[i].reshape(28, 28), cmap='gray')
  plt.title(f'Example image {i+1}')
  plt.axis('off')
plt.suptitle('Original Images of Digit 3')
plt.show()
```

Original Images of Digit 3



[269]:

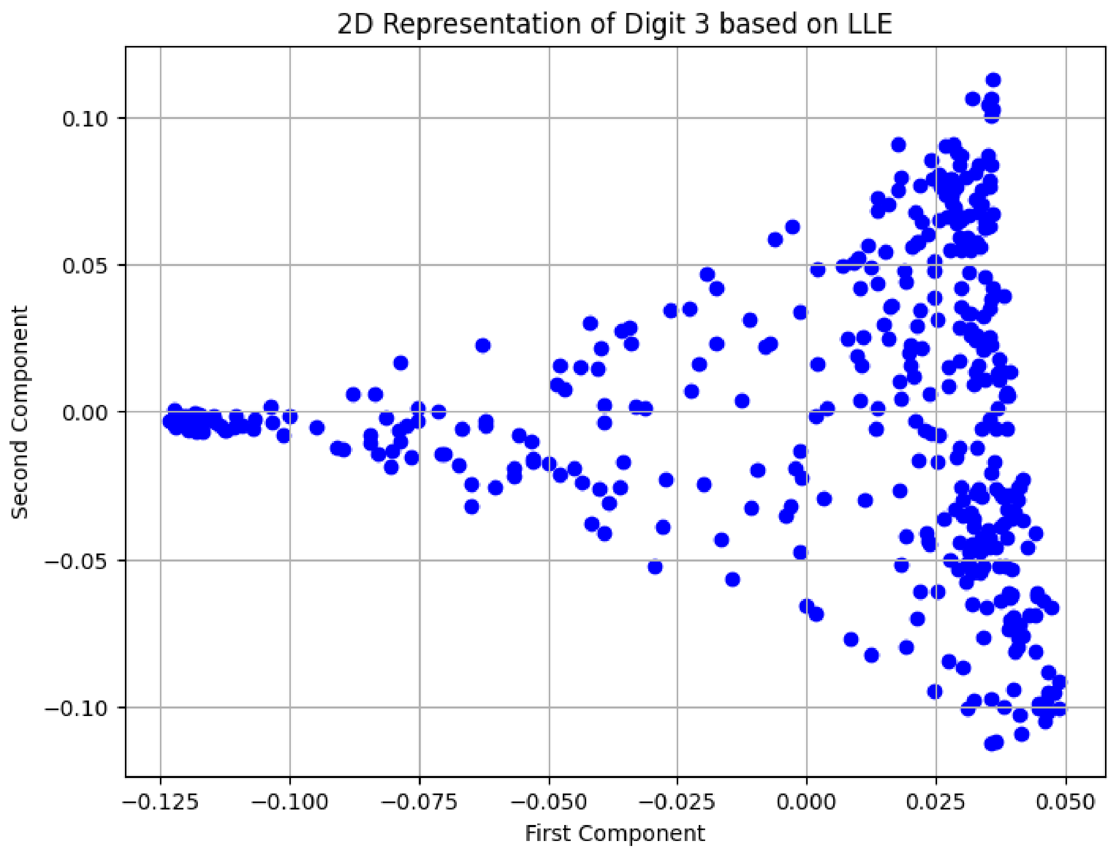
```
neighbours = 5
low_dimension = 4
```

[270]:

```
lle = LocallyLinearEmbedding(n_neighbors = neighbours, n_components = 2)

x_lle = lle.fit_transform(x)
```

[271]:

```
plt.figure(figsize=(8, 6))
plt.scatter(x_lle[:, 0], x_lle[:, 1], c='blue',marker='o')
plt.title('2D Representation of Digit 3 based on LLE')
plt.xlabel('First Component')
plt.ylabel('Second Component')
plt.grid(True)
plt.show()
```



2 Analysis

2.0.1 The data points are dispersed primarily along the first component (horizontal axis); suggesting that this component captures the most significant variation within the data.

2.0.2 The second component captures the next most significant variation orthogonal to the first.

2.0.3 We dont see distinct clusters, but there are regions where the points are more sparse. There could be potential outliers, on the fringes of the spread along the first component.

[ ]:

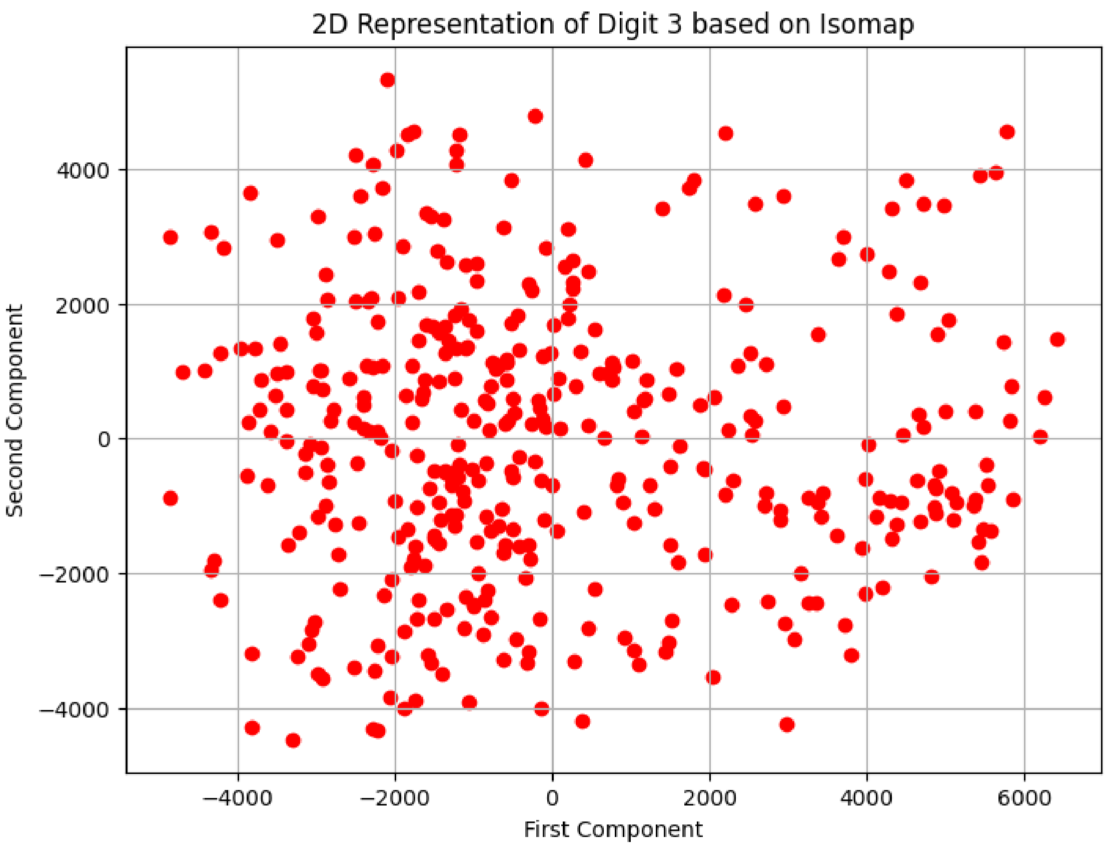
2.0.4 Q2 Repeat step 1 using the ISOMAP method. Comment on the result. Does ISOMAP do better in some way? Are the patterns being found globally based or locally based?

[272]:

```
isomap = Isomap(n_neighbors=neighbours, n_components = 2)

x_iso = isomap.fit_transform(x)
```

[273]:

```
plt.figure(figsize=(8, 6))
plt.scatter(x_iso[:, 0], x_iso[:, 1], c='red',marker='o')
plt.title('2D Representation of Digit 3 based on Isomap')
plt.xlabel('First Component')
plt.ylabel('Second Component')
plt.grid(True)
plt.show()
```



3 Analysis

3.0.1 In the Isomap Plot, the points are dispersed but they have a higher degree of clustering as several distinct clusters are visible.

3.0.2 It tends to preserve the global geometry and geodesic distances between points. The points that are far apart in the original space tend to be far apart in the reduced space as well.

3.0.3 Isomap is better at capturing global relationships and could be more suitable for tasks like clustering or visualization. While LLE is senesitive to local vairations in the data and capture finer details, which might be lost in global approach like Isomap. For clustering purpose ISOMAP is better than LLE.

3.0.4 In terms of pattern found, ISOMAP captures globally based since it aims to preserve the global geometry of the data.

[ ]:

```

```

3.0.5 Q3 Use the Naive Bayes classifier to classify the dataset based on the projected 4-dimension representations of the LLE and ISOMAP. Train your classifier by randomly selected 70% of data, and test with remained 30%. Retrain for multiple iterations (using different random partitions of the data) and use the average accuracy of multiple runs for your analysis. Justify why your number of iterations was sufficient. Based on the average accuracies compare their performance with PCA and LDA. Discuss the result.

[274]:

```
data = pd.read_csv('DataC.csv')

x = data.iloc[:,:-1].values
y = data.iloc[:,-1].values
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

[275]:

```
def perform_analysis(data, labels, n_components=4, iterations=10):  # Reduced␣
↪for testing
    results = {
        'LLE': [],
        'ISOMAP': [],
        'PCA': [],
        'LDA': []
    }

    for _ in range(iterations):
        print("Currently on iteration: ", _+1)
        X_train, X_test, y_train, y_test = train_test_split(data, labels,␣
↪test_size=0.3,random_state=42)
```

```
        lle = LocallyLinearEmbedding(n_neighbors=5,n_components=n_components)
        isomap = Isomap(n_neighbors=5,n_components=n_components)
        pca = PCA(n_components=n_components)
        lda = LinearDiscriminantAnalysis(n_components=n_components)

        for method, transformer in zip(['LLE', 'ISOMAP', 'PCA', 'LDA'], [lle,␣
↪isomap, pca, lda]):
            X_train_transformed = transformer.fit_transform(X_train, y_train)
            X_test_transformed = transformer.transform(X_test)

            classifier = GaussianNB()
            classifier.fit(X_train_transformed, y_train)
            predictions = classifier.predict(X_test_transformed)
            accuracy = accuracy_score(y_test, predictions)
            results[method].append(accuracy*100)

    average_accuracies = {method: np.mean(accuracies) for method, accuracies in␣
↪results.items()}
    return average_accuracies

average_accuracies = perform_analysis(x, y)
print(average_accuracies)
```

```
Currently on iteration:  1
Currently on iteration:  2
Currently on iteration:  3
Currently on iteration:  4
Currently on iteration:  5
Currently on iteration:  6
Currently on iteration:  7
Currently on iteration:  8
Currently on iteration:  9
Currently on iteration:  10
{'LLE': 98.22580645161291, 'ISOMAP': 95.32258064516131, 'PCA':
92.74193548387096, 'LDA': 94.67741935483869}
```

4 Analysis

4.0.1 The number of iterations (25) used to train and test the model. With two few iterations the accuracy estimate might be unreliable and susceptible to the variance introduced by random partitioning of the dataset. With too many it might be computationally tough to train.

4.0.2 From the accuracy results, LLE has the highest accuracy, indicating the local information captured by LLE projection is informative for the Naive Bayes classifier. It has preserved essential features.

4.0.3 ISOMAP performs well but not as effective as LLE in this experiment. This suggests the global properties preserved by ISOMAP are less essential than the properties captured by LLE.

4.0.4 PCA and LDA have lower accuracies than LLE, while PCA is least accurate. The lower accuracy of PCA suggests that variance-based approach to dimensionality reduction may not capture the most relevant features for classification in this case. LDA does slightly better than PCA, as LDA is designed to maximize the class sperability.

assignment2-binary-classification

March 26, 2024

1 Binary Classification

1.0.1 Classify data set A1 using four classifiers: k-NN, Support Vector Machine (with rbf kernel), Naïve Bayes Classifier, and Decision Tree. The objective is to experiment with parameter selection in training classifiers and to compare the performance of these well-known classification methods.

[25]:
```
#Importing Necessary Libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split,␣
↪GridSearchCV,cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from statistics import mean, stdev
from matplotlib.ticker import MultipleLocator
from scipy.stats import norm
```

[26]:
```
# Loading DataA1
DataA1 = pd.read_csv('DataA1.csv',encoding='latin-1')
DataA1.head()
```

[26]:
```
   Feature1  Feature2  Feature3  Feature4  Feature5  Feature6  Feature7  \
0         1         2         1         2         1         2         3
1         3         3         4         2         1         2         2
2         4         1         4         4         4         4         1
3         1         4         1         1         3         3         4
4         3         4         4         3         1         1         4

   Feature8  Feature9  Feature10  …  Feature49  Feature50  Feature51  \
0         3         3          3  …          3          2          3
```

```
1         4         3          2  …          1          4          3
2         1         2          1  …          1          2          1
3         4         3          4  …          1          3          3
4         4         4          1  …          3          1          3

   Feature52  Feature53  Feature54  Feature55  Feature56  Feature57  Label
0          4          2          2          2          2          1      1
1          4          4          4          1          3          4      1
2          1          4          2          2          4          4      1
3          4          1          3          3          4          2     -1
4          2          1          4          2          1          1     -1

[5 rows x 58 columns]
```

[27]:
```
#Print the columns
print(DataA1.columns)
```

```
Index(['Feature1', 'Feature2', 'Feature3', 'Feature4', 'Feature5', 'Feature6',
       'Feature7', 'Feature8', 'Feature9', 'Feature10', 'Feature11',
       'Feature12', 'Feature13', 'Feature14', 'Feature15', 'Feature16',
       'Feature17', 'Feature18', 'Feature19', 'Feature20', 'Feature21',
       'Feature22', 'Feature23', 'Feature24', 'Feature25', 'Feature26',
       'Feature27', 'Feature28', 'Feature29', 'Feature30', 'Feature31',
       'Feature32', 'Feature33', 'Feature34', 'Feature35', 'Feature36',
       'Feature37', 'Feature38', 'Feature39', 'Feature40', 'Feature41',
       'Feature42', 'Feature43', 'Feature44', 'Feature45', 'Feature46',
       'Feature47', 'Feature48', 'Feature49', 'Feature50', 'Feature51',
       'Feature52', 'Feature53', 'Feature54', 'Feature55', 'Feature56',
       'Feature57', 'Label'],
      dtype='object')
```

[28]:
```
#Get the shape of the dataset
DataA1.shape
```

[28]: (2200, 58)

[29]:
```
#Get top 5 rows of the dataset
DataA1.head()
```

[29]:
```
   Feature1  Feature2  Feature3  Feature4  Feature5  Feature6  Feature7  \
0         1         2         1         2         1         2         3
1         3         3         4         2         1         2         2
2         4         1         4         4         4         4         1
3         1         4         1         1         3         3         4
4         3         4         4         3         1         1         4

   Feature8  Feature9  Feature10  …  Feature49  Feature50  Feature51  \
```

```
0         3         3         3  ...         3         2         3
1         4         3         2  ...         1         4         3
2         1         2         1  ...         1         2         1
3         4         3         4  ...         1         3         3
4         4         4         1  ...         3         1         3

   Feature52  Feature53  Feature54  Feature55  Feature56  Feature57  Label
0          4          2          2          2          2          1      1
1          4          4          4          1          3          4      1
2          1          4          2          2          4          4      1
3          4          1          3          3          4          2     -1
4          2          1          4          2          1          1     -1

[5 rows x 58 columns]
```

[30]:

```
# Knowing the dataset
# Checking the non-null and null values for each columns
DataA1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2200 entries, 0 to 2199
Data columns (total 58 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Feature1   2200 non-null   int64
 1   Feature2   2200 non-null   int64
 2   Feature3   2200 non-null   int64
 3   Feature4   2200 non-null   int64
 4   Feature5   2200 non-null   int64
 5   Feature6   2200 non-null   int64
 6   Feature7   2200 non-null   int64
 7   Feature8   2200 non-null   int64
 8   Feature9   2200 non-null   int64
 9   Feature10  2200 non-null   int64
 10  Feature11  2200 non-null   int64
 11  Feature12  2200 non-null   int64
 12  Feature13  2200 non-null   int64
 13  Feature14  2200 non-null   int64
 14  Feature15  2200 non-null   int64
 15  Feature16  2200 non-null   int64
 16  Feature17  2200 non-null   int64
 17  Feature18  2200 non-null   int64
 18  Feature19  2200 non-null   int64
 19  Feature20  2200 non-null   int64
 20  Feature21  2200 non-null   int64
 21  Feature22  2200 non-null   int64
 22  Feature23  2200 non-null   int64
```

```
 23  Feature24  2200 non-null   int64
 24  Feature25  2200 non-null   int64
 25  Feature26  2200 non-null   int64
 26  Feature27  2200 non-null   int64
 27  Feature28  2200 non-null   int64
 28  Feature29  2200 non-null   int64
 29  Feature30  2200 non-null   int64
 30  Feature31  2200 non-null   int64
 31  Feature32  2200 non-null   int64
 32  Feature33  2200 non-null   int64
 33  Feature34  2200 non-null   int64
 34  Feature35  2200 non-null   int64
 35  Feature36  2200 non-null   int64
 36  Feature37  2200 non-null   int64
 37  Feature38  2200 non-null   int64
 38  Feature39  2200 non-null   int64
 39  Feature40  2200 non-null   int64
 40  Feature41  2200 non-null   int64
 41  Feature42  2200 non-null   int64
 42  Feature43  2200 non-null   int64
 43  Feature44  2200 non-null   int64
 44  Feature45  2200 non-null   int64
 45  Feature46  2200 non-null   int64
 46  Feature47  2200 non-null   int64
 47  Feature48  2200 non-null   int64
 48  Feature49  2200 non-null   int64
 49  Feature50  2200 non-null   int64
 50  Feature51  2200 non-null   int64
 51  Feature52  2200 non-null   int64
 52  Feature53  2200 non-null   int64
 53  Feature54  2200 non-null   int64
 54  Feature55  2200 non-null   int64
 55  Feature56  2200 non-null   int64
 56  Feature57  2200 non-null   int64
 57  Label      2200 non-null   int64
dtypes: int64(58)
memory usage: 997.0 KB
```

[31]:

```
# Separating features and Target Attribute (Label)
x = DataA1.drop(columns=['Label'])
y = DataA1['Label']
x.head()
```

[31]:

```
   Feature1  Feature2  Feature3  Feature4  Feature5  Feature6  Feature7  \
0         1         2         1         2         1         2         3
1         3         3         4         2         1         2         2
2         4         1         4         4         4         4         1
```

```
3          1          4          1          1          3          3          4
4          3          4          4          3          1          1          4

   Feature8  Feature9  Feature10  …  Feature48  Feature49  Feature50  \
0         3         3          3  …          4          3          2
1         4         3          2  …          3          1          4
2         1         2          1  …          1          1          2
3         4         3          4  …          2          1          3
4         4         4          1  …          4          3          1

   Feature51  Feature52  Feature53  Feature54  Feature55  Feature56  Feature57
0          3          4          2          2          2          2          1
1          3          4          4          4          1          3          4
2          1          1          4          2          2          4          4
3          3          4          1          3          3          4          2
4          3          2          1          4          2          1          1

[5 rows x 57 columns]
```

[32]:
```python
y.head()
```

[32]:
```
0    1
1    1
2    1
3   -1
4   -1
Name: Label, dtype: int64
```

[33]:
```python
# Z-score normalization
scaler = StandardScaler()
normalised_x = scaler.fit_transform(x)
np.array(normalised_x)
```

[33]:
```
array([[-1.35289759, -0.48747864, -1.37244139, …, -0.4584159 ,
        -0.41756618, -1.39224875],
       [ 0.45920268,  0.43308188,  1.36002111, …, -1.37441497,
         0.49741947,  1.36965138],
       [ 1.36525282, -1.40803915,  1.36002111, …, -0.4584159 ,
         1.41240513,  1.36965138],
       …,
       [ 0.45920268,  1.35364239,  1.36002111, …,  0.45758317,
        -1.33255183, -0.47161537],
       [-1.35289759,  1.35364239,  0.44920027, …,  1.37358225,
         0.49741947,  1.36965138],
       [ 1.36525282,  0.43308188, -1.37244139, …, -0.4584159 ,
         1.41240513,  0.44901801]])
```

[34]:
```python
# Split data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(normalised_x, y,␣
↪test_size=0.3, random_state=42)
np.array(x_train).shape
```

[34]: (1540, 57)

[35]:
```python
np.array(x_test).shape
```

[35]: (660, 57)

[36]:
```python
np.array(y_train).shape
```

[36]: (1540,)

[37]:
```python
np.array(y_test).shape
```

[37]: (660,)

2. Use 5-fold cross validation on the training set to select the parameters k for k-NN from the set [1, 3, 5, 7, ..., 31]. Plot a figure that shows the relationship between the accuracy and the parameter k. Report the best k in terms of classification accuracy.

[38]:
```python
# Possible Values for k
k = np.arange(1, 32, 2)
print("k values: \n",k)

# Perform 5-fold cross-validation to select k
cross_validation_scores = []
for k_value in k:
    K_nearest_neighbour = KNeighborsClassifier(n_neighbors=k_value)
    validation_scores = cross_val_score(K_nearest_neighbour, x_train, y_train,␣
↪cv = 5, scoring='accuracy')

    cross_validation_scores.append(validation_scores.mean())

    print("k value:",k_value,"\nAccuracy Scores", validation_scores, "\nMean␣
↪Value of Accuracy: ", validation_scores.mean(),"\n")
```

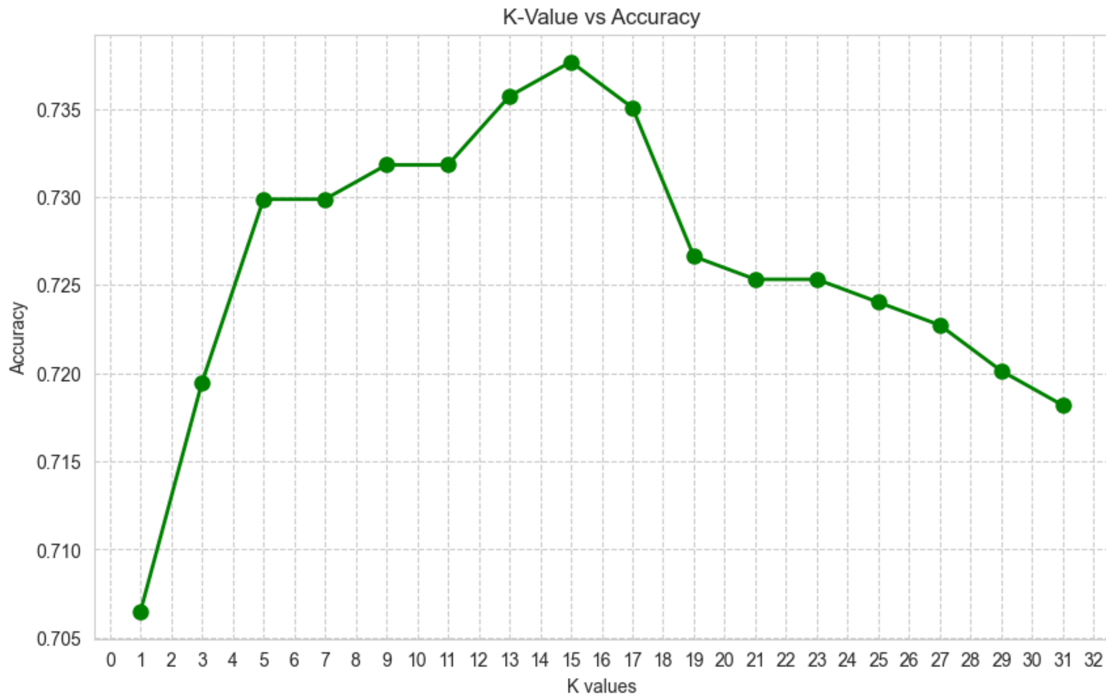
```
k values:
 [ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31]
k value: 1
Accuracy Scores [0.69805195 0.69805195 0.69155844 0.72077922 0.72402597]
Mean Value of Accuracy:  0.7064935064935064

k value: 3
Accuracy Scores [0.7012987  0.73376623 0.70779221 0.73701299 0.71753247]
Mean Value of Accuracy:  0.7194805194805195
```

```
k value: 5
Accuracy Scores [0.72727273 0.73376623 0.75324675 0.72402597 0.71103896]
Mean Value of Accuracy:  0.7298701298701299

k value: 7
Accuracy Scores [0.73701299 0.74675325 0.73376623 0.73051948 0.7012987 ]
Mean Value of Accuracy:  0.7298701298701298

k value: 9
Accuracy Scores [0.74675325 0.75324675 0.73701299 0.72402597 0.69805195]
Mean Value of Accuracy:  0.7318181818181817

k value: 11
Accuracy Scores [0.75649351 0.76298701 0.71103896 0.73051948 0.69805195]
Mean Value of Accuracy:  0.7318181818181817

k value: 13
Accuracy Scores [0.73701299 0.77272727 0.71103896 0.74350649 0.71428571]
Mean Value of Accuracy:  0.7357142857142858

k value: 15
Accuracy Scores [0.72727273 0.75324675 0.73376623 0.75324675 0.72077922]
Mean Value of Accuracy:  0.7376623376623377

k value: 17
Accuracy Scores [0.72077922 0.75       0.72727273 0.76298701 0.71428571]
Mean Value of Accuracy:  0.7350649350649351

k value: 19
Accuracy Scores [0.72077922 0.73701299 0.72077922 0.75       0.70454545]
Mean Value of Accuracy:  0.7266233766233767

k value: 21
Accuracy Scores [0.71103896 0.75       0.70779221 0.75974026 0.69805195]
Mean Value of Accuracy:  0.7253246753246754

k value: 23
Accuracy Scores [0.68831169 0.75       0.71103896 0.76623377 0.71103896]
Mean Value of Accuracy:  0.7253246753246754

k value: 25
Accuracy Scores [0.69155844 0.75974026 0.71428571 0.74350649 0.71103896]
Mean Value of Accuracy:  0.724025974025974

k value: 27
Accuracy Scores [0.69155844 0.76298701 0.71753247 0.74025974 0.7012987 ]
Mean Value of Accuracy:  0.7227272727272728
```

```
k value: 29
Accuracy Scores [0.69805195 0.74025974 0.71428571 0.73701299 0.71103896]
Mean Value of Accuracy:  0.7201298701298702

k value: 31
Accuracy Scores [0.69805195 0.74350649 0.70454545 0.73376623 0.71103896]
Mean Value of Accuracy:  0.7181818181818181
```

[39]:

```python
# Plot K-value vs Accuracy
# Set Seaborn style
sns.set_style("whitegrid")

# Create a new figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the line plot using Matplotlib with markers
ax.plot(k, cross_validation_scores, marker='o', markersize=8, linestyle='-',␣
↪color='Green', linewidth=2)

# Set labels and title
ax.set_xlabel('K values')
ax.set_ylabel('Accuracy')
ax.set_title('K-Value vs Accuracy')

# Set axis units to 1 point
ax.xaxis.set_major_locator(MultipleLocator(1))

# Set additional visual enhancements
ax.grid(True, linestyle='--', alpha=1)

# Show plot
plt.show()
```



[40]:
```
# Find best K-value
best_k = k[np.argmax(cross_validation_scores)]
print("Best Value of K for k-NN Classifier is:", best_k)
```

```
Best Value of K for k-NN Classifier is: 15
```

3. For the RBF kernel SVM, there are two parameters to be decided: the soft margin penalty term c and the kernel width parameter gamma. Again use 5-fold cross validation on the training set to select the parameter c from the set [0.1, 0.5, 1, 2, 5, 10, 20, 50] and select the parameter gamma from the set [0.01, 0.05, 0.1, 0.5, 1, 2, 5, 10]. Report the best parameters in terms of classification accuracy.

[41]:
```
# Define candidate values for parameters c and gamma
parameters = {'C': [0.1, 0.5, 1, 2, 5, 10, 20, 50], 'gamma': [0.01, 0.05, 0.1,␣
↪0.5, 1, 2, 5, 10]}

# Perform grid search with 5-fold cross-validation
svm_grid = GridSearchCV(SVC(kernel='rbf'), parameters, cv=5, scoring='accuracy')
svm_grid.fit(x_train, y_train)

# Get best parameters
best_c = svm_grid.best_params_['C']
best_gamma = svm_grid.best_params_['gamma']
print("Best parameters for SVM (RBF kernel): C =", best_c, ", gamma =",␣
↪best_gamma)
```

```
Best parameters for SVM (RBF kernel): C = 10 , gamma = 0.01
```

4. Using the chosen parameters from the above parameter selection process for k-NN and SVM, and the default setups for Naïve Bayes classifier and Decision Tree, classify the test set. Repeat each classification method 20 times by varying the split of training-test set as in Step (1). Report the average and standard deviation of classification performance on the test set regarding accuracy, precision, recall, and F1-score.

[42]:

```python
# Function to perform classification and return performance metrics
def classify_and_evaluate(classifier, X_train, y_train, X_test, y_test):
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    return accuracy, precision, recall, f1
```

[43]:

```python
# Initialize classifiers with selected parameters
knn = KNeighborsClassifier(n_neighbors=best_k)
svm = SVC(kernel='rbf', C = best_c, gamma = best_gamma)
naive_bayes = GaussianNB()
decision_tree = DecisionTreeClassifier()
```

[44]:

```python
# Perform classification and evaluation 20 times
def Perform_classification(classifier):
    num_trials = 20
    accuracy_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []

    for _ in range(num_trials):
        # Split data into train and test sets
        x_train, x_test, y_train, y_test = train_test_split(normalised_x, y,␣
↪test_size=0.3, random_state=None)

        # Classify and evaluate
        accuracy, precision, recall, f1 = classify_and_evaluate(classifier,␣
↪x_train, y_train, x_test, y_test)
        accuracy_scores.append(accuracy)
        precision_scores.append(precision)
        recall_scores.append(recall)
        f1_scores.append(f1)

    # Report average and standard deviation of performance metrics
    print("Average Accuracy of:", mean(accuracy_scores))
```

```
    print("Standard Deviation Accuracy:", stdev(accuracy_scores))
    print("Average Precision:", mean(precision_scores))
    print("Standard Deviation Precision:", stdev(precision_scores))
    print("Average Recall:", mean(recall_scores))
    print("Standard Deviation Recall:", stdev(recall_scores))
    print("Average F1-score:", mean(f1_scores))
    print("Standard Deviation F1-score:", stdev(f1_scores))
```

[45]:
```
# KNN
Perform_classification(knn)
```

```
Average Accuracy of: 0.7454545454545455
Standard Deviation Accuracy: 0.023062388461276794
Average Precision: 0.9582317038584496
Standard Deviation Precision: 0.017951948079664548
Average Recall: 0.5281739994932426
Standard Deviation Recall: 0.038444546032150424
Average F1-score: 0.6799594992015696
Standard Deviation F1-score: 0.03055886501181418
```

[46]:
```
# SVM Perform classification and evaluation 20 times
Perform_classification(svm)
```

```
Average Accuracy of: 0.9056818181818181
Standard Deviation Accuracy: 0.008512642447346968
Average Precision: 0.9229803087200773
Standard Deviation Precision: 0.012422421195910737
Average Recall: 0.8888911185738743
Standard Deviation Recall: 0.015071708266128764
Average F1-score: 0.9055140347602055
Standard Deviation F1-score: 0.009756328843081872
```

[47]:
```
# Naiye Bayes Perform classification and evaluation 20 times
Perform_classification(naive_bayes)
```

```
Average Accuracy of: 0.870530303030303
Standard Deviation Accuracy: 0.01231381077168416
Average Precision: 0.8695971015077553
Standard Deviation Precision: 0.01612655811730161
Average Recall: 0.8837044306019265
Standard Deviation Recall: 0.014578520189126608
Average F1-score: 0.8764875415994177
Standard Deviation F1-score: 0.011734367684576799
```

[48]:
```
# Decision Tree Perform classification and evaluation 20 times
Perform_classification(decision_tree)
```

```
Average Accuracy of: 0.9343181818181818
```

```
Standard Deviation Accuracy: 0.011800737544827626
Average Precision: 0.9394914827870673
Standard Deviation Precision: 0.014165389994089742
Average Recall: 0.9328387152109977
Standard Deviation Recall: 0.016293521724058575
Average F1-score: 0.9360564593071852
Standard Deviation F1-score: 0.011767181761779988
```

5 Comment on the obtained results.

2 Analysis

2.0.1 KNN classifier shows a moderate level of accuracy. The high precision suggests that model is likely to predict a positive class correct. The low recall indicates model misses a significant number of actual positive cases. There is a tradeoff between recall and precision. The F1-score is also low , thus it does not capture the complexity of the data.

2.0.2 SVM with RBF Kernel demonstrates strong performace across all metrices with high accuracy, precision, recall and F1-score. The low standard deviation indicates model is consisten across whole data and chosen parameters are effective.

2.0.3 Naive Bayes Classifier has strong F1-score. With balanced precision and recall , Naive Bayes looks good for this dataset.

2.0.4 Decision Tree shows highest accuracy among all the classifiers and has a high F1-score as well; indicating good balance between precision and recall. The low standard deviation suggests that it is stable across different data splits.

a2-multiclass-classification

March 26, 2024

1 Multi-Class Classification

[576]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier
from sklearn.metrics import accuracy_score, precision_score, f1_score,␣
    ↪confusion_matrix, recall_score
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler
```

[577]:
```
data = pd.read_csv('DataB1.csv')
```

[578]:
```
data.head()
```

[578]:

	sepallength	sepalwidth	petallength	petalwidth	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

[579]:
```
data.columns
```

[579]:
```
Index(['sepallength', 'sepalwidth', 'petallength', 'petalwidth', 'class'],
dtype='object')
```

[580]:
```
data.shape
```

[580]:
```
(150, 5)
```

[581]:
```
data.isnull().sum()
```

[581]:
```
sepallength    0
sepalwidth     0
```

```
petallength    0
petalwidth     0
class          0
dtype: int64
```

[582]:
```
# Assuming the last column is the label
X = data.drop(columns=['class'])
y = data['class']
```

[583]:
```
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)
```

[584]:
```
# Encode class labels to integers
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
```

[585]:
```
# Splits the dataset into training (70%) and test (30%) sets.
X_train, X_test, y_train, y_test = train_test_split(X_normalized, y_encoded,␣
↪test_size=0.3, random_state=42)
```

2 1. Describe and develop the training and classification procedures by using the "one-versus-all" and "one-versus-one" strategy for SVM.

One-versus-One (OvO) Strategy for SVM

- Description: In the one-versus-one (OvO) strategy, a binary classifier is trained for every pair of classes. If there are N classes, the results in $\frac{N\times(N-1)}{2}$ classifiers.
- Procedure:
  1. Training: Train a binary SVM classifier for each pair of classes.
  2. Classification: To classify a new sample, run it through all classifiers. The class that wins the most duels (is chosen most frequently by the classifiers) is assigned to the sample.

One-versus-All (OvA) Strategy for SVM

- Description: In the one-versus-all (OvA) strategy, a single classifier is trained per class to distinguish the samples of that class from samples of all other classes. For a problem with N classes, N separate binary classifiers are trained. For each classifier, the class it represents is treated as a positive class, while all other classes are merged into a negative class.
- Procedure:
  1. Training: For each class i, train a binary SVM classifier where class i samples are considered positive, and all other samples are considered negative.
  2. Classification: To classify a new sample, run it through all N classifiers. The classifier that outputs the highest confidence score (distance from the decision boundary) assigns its class to the sample.

3 2. Classify data set B by using binary Support Vector Machine classifiers with linear kernel and default parameters. Randomly split the data into 70% training and 30% test set. Report the classification overall accuracy, precision, recall, F1-score, and the confusion matrix of the classification results on the test set.

[586]:
```
# Initialize the SVM One-vs-One classifier with a linear kernel
model_one_one = SVC(decision_function_shape='ovo', kernel = 'linear')
# Train the classifier
model_one_one.fit(X_train, y_train)
# Predict on the test set
y_pred_one_one = model_one_one.predict(X_test)

# Calculate evaluation metrics
accuracy_one_one = accuracy_score(y_test, y_pred_one_one)
precision_one_one = precision_score(y_test, y_pred_one_one, average='weighted')
recall_one_one = recall_score(y_test, y_pred_one_one, average='weighted')
f1_one_one = f1_score(y_test, y_pred_one_one, average='weighted')
conf_matrix_one_one = confusion_matrix(y_test, y_pred_one_one)
```

[587]:
```
# Initialize the SVM One-vs-All classifier with a linear kernel
model_one_all = SVC(decision_function_shape='ovr', kernel = 'linear')
# Train the classifier
model_one_all.fit(X_train, y_train)
# Predict on the test set
y_pred_one_all = model_one_all.predict(X_test)

# Calculate evaluation metrics
accuracy_one_all = accuracy_score(y_test, y_pred_one_all)
precision_one_all = precision_score(y_test, y_pred_one_all, average='weighted')
recall_one_all = recall_score(y_test, y_pred_one_all, average='weighted')
f1_one_all = f1_score(y_test, y_pred_one_all, average='weighted')
conf_matrix_one_all = confusion_matrix(y_test, y_pred_one_all)
```

[588]:
```
print("One-vs-One:")
print(f"Accuracy for one-versus-one: {accuracy_one_one}")
print(f"Precision for one-versus-one: {precision_one_one}")
print(f"Recall for one-versus-one: {recall_one_one}")
print(f"F1-Score for one-versus-one: {f1_one_one}")
print("Confusion Matrix for one-versus-one:")
print(conf_matrix_one_one)
```

```
One-vs-One:
Accuracy for one-versus-one: 0.9777777777777777
Precision for one-versus-one: 0.9793650793650793
Recall for one-versus-one: 0.9777777777777777
```

```
F1-Score for one-versus-one: 0.9777448559670783
Confusion Matrix for one-versus-one:
[[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
```

[589]:
```
print("One-vs-All:")
print(f"Accuracy for one-versus-all: {accuracy_one_all}")
print(f"Precision for one-versus-all: {precision_one_all}")
print(f"Recall for one-versus-all: {recall_one_all}")
print(f"F1-Score for one-versus-all: {f1_one_all}")
print("Confusion Matrix for one-versus-all:")
print(conf_matrix_one_all)
```

```
One-vs-All:
Accuracy for one-versus-all: 0.9777777777777777
Precision for one-versus-all: 0.9793650793650793
Recall for one-versus-all: 0.9777777777777777
F1-Score for one-versus-all: 0.9777448559670783
Confusion Matrix for one-versus-all:
[[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
```

When using a Support Vector Machine (SVM) with a 'linear' kernel or without specifying kernel settings, choosing between One-vs-One (OvO) and One-vs-All (OvA) strategies can result in different classification outcomes due to their distinct approaches to handling multi-class classification

[590]:
```
# Initialize the SVM classifier with linear kernel and default parameters
svm_classifier = SVC(kernel='linear')

# Train the SVM classifier on the training data
svm_classifier.fit(X_train, y_train)

# Predict labels for the test set
y_pred = svm_classifier.predict(X_test)
```

[591]:
```
# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)


print(f"SVM classifier Accuracy: {accuracy}")
print(f"SVM classifier Precision: {precision}")
print(f"SVM classifier Recall: {recall}")
```

```
print(f"SVM classifier F1-Score: {f1}")
print("SVM classifier Confusion Matrix:")
print(conf_matrix)
```

```
SVM classifier Accuracy: 0.9777777777777777
SVM classifier Precision: 0.9793650793650793
SVM classifier Recall: 0.9777777777777777
SVM classifier F1-Score: 0.9777448559670783
SVM classifier Confusion Matrix:
[[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
```

The results for One-vs-One and One-vs-All strategies mentioned above are based on a normalized dataset. Without data normalization, the outputs for both strategies could differ due to variations in feature scales.

The effectiveness of One-vs-One and One-vs-All strategies in SVM classification can be significantly influenced by whether the dataset is normalized. Without normalization, the disparities in scale among features can lead to different outcomes for both strategies, potentially affecting their performance and accuracy.

4 3. How does the decision tree classifier deal with the multi-class problem? Classify data set B using decision tree with default parameters, report the classification results. Comment and compare the methods of SVM and decision tree.

For Decision Tree classifiers, both One-vs-One (OvO) and One-vs-All (OvA) strategies can be applied:

- **OvO**: A binary decision tree is created for each class pair, each trained to differentiate between its two classes. The final prediction is made based on the majority vote from all trees.
- **OvA**: A binary decision tree is created for each class to distinguish it from all others. The final prediction is the class that a decision tree identifies with the highest confidence.

A Decision Tree classifier handles multi-class problems directly by:

- **Natively Supporting Multi-Class**: Can classify instances into multiple classes without needing special strategies.
- **Hierarchical Splitting**: Uses feature values to recursively split the data, creating branches and leaves that correspond to different classes.
- **Purity-Based Decisions**: Chooses splits based on measures like Gini impurity or entropy to best separate classes.
- **Binary Splits for Multi-Class**: Although splits are binary, the hierarchical nature allows for efficient multi-class classification.
- **Leaf Nodes Represent Classes**: Each leaf node predicts a class, guiding instances to the appropriate class based on their features.

[592]:
```
# Splits the dataset into training (70%) and test (30%) sets without feature
↪scaling
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.
↪3, random_state=42)
```

[593]:
```
# Initialize the Decision Tree classifier
dt_classifier  = DecisionTreeClassifier()

# Train the classifier
dt_classifier  = dt_classifier .fit(X_train, y_train)

# Predict on the test se
y_pred_dt  = dt_classifier .predict(X_test)
```

[594]:
```
# Calculate evaluation metrics
accuracy_dt = accuracy_score(y_test, y_pred_dt )
precision_dt = precision_score(y_test, y_pred_dt , average='weighted')
recall_dt = recall_score(y_test, y_pred_dt , average='weighted')
f1_dt = f1_score(y_test, y_pred_dt , average='weighted')
conf_matrix_dt = confusion_matrix(y_test, y_pred_dt )


print(f"Decision Tree Accuracy: {accuracy_dt}")
print(f"Decision Tree Precision: {precision_dt}")
print(f"Decision Tree Recall: {recall_dt}")
print(f"Decision Tree F1 Score: {f1_dt}")
print(f"Decision Tree Confusion Matrix:\n{conf_matrix_dt}")
```

```
Decision Tree Accuracy: 1.0
Decision Tree Precision: 1.0
Decision Tree Recall: 1.0
Decision Tree F1 Score: 1.0
Decision Tree Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

- **Multi-Class Handling**:
  - **SVM**: Uses One-versus-All or One-versus-One strategies for multi-class classification, increasing complexity.
  - **Decision Tree**: Natively supports multi-class classification without additional complexity.
- **Performance**:
  - **SVM**: Often excels in precision and recall for linearly separable data, with the ability to handle high-dimensional spaces effectively.
  - **Decision Tree**: Can capture complex patterns but may overfit, affecting precision and recall.
- **Interpretability**:

  - **SVM**: Less intuitive due to abstract concepts like hyperplanes and support vectors.
  - **Decision Tree**: Highly interpretable with a clear visualization of decision paths.
- **Model Complexity and Overfitting**:
  - **SVM**: Requires careful selection of kernel and regularization to balance bias and variance.
  - **Decision Tree**: Prone to overfitting, especially with deep trees; requires pruning or constraints to ensure generalizability.

Overall, the choice between SVM and Decision Trees depends on the specific needs for interpretability, the nature of the dataset, and the trade-off between performance and model complexity.