ml-200090107072-1

October 20, 2023

- Name : Raval Meet
- Enrollment No : 200090107072
- Div : A

0.1 CO-1 ASSIGNMENT:

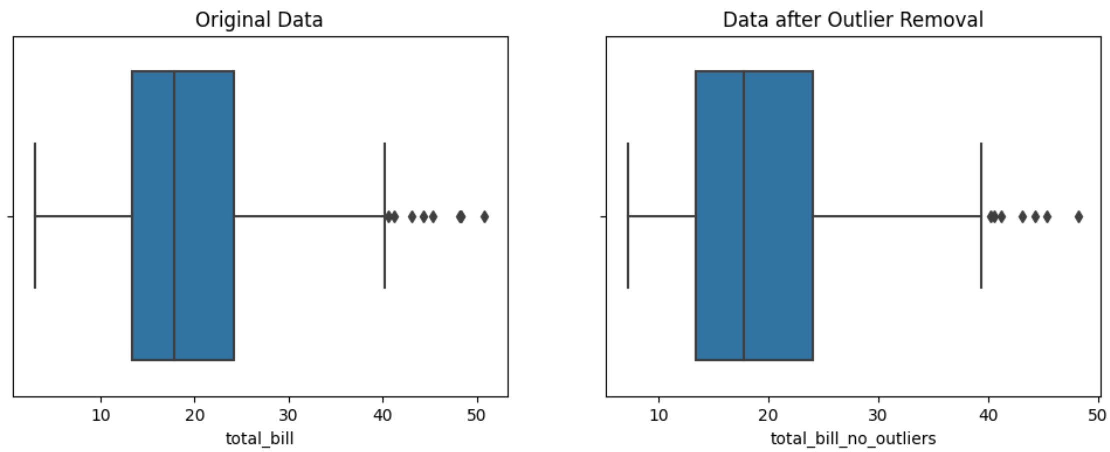
1. Implement the techniques to deal with outliers. - https://www.analyticsvidhya.com/blog/2021/05/feature-engineering-how-to-detect-and-remove-outliers-with-python-code/

[8]:
```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Load the dataset (you can choose any built-in dataset from Seaborn)
# We'll use the 'tips' dataset for this example
data = sns.load_dataset('tips')

# Display the first few rows of the dataset
print("Original Data:")
print(data.head())

# Define a function to visualize the data and outliers
def plot_with_outliers(data, column_name):
    plt.figure(figsize=(12, 4))

    # Plot the original data
    plt.subplot(1, 2, 1)
    sns.boxplot(x=data[column_name])
    plt.title("Original Data")

    # Plot the data after outlier removal
    plt.subplot(1, 2, 2)
    sns.boxplot(x=data[column_name+'_no_outliers'])
    plt.title("Data after Outlier Removal")

    plt.show()
```

```python
# Method 1: Z-score Treatment
def z_score_outlier_treatment(data, column_name):
    from scipy import stats

    # Calculate Z-scores for the column
    z_scores = np.abs(stats.zscore(data[column_name]))

    # Define a threshold for considering data as outliers (e.g., Z-score > 3)
    threshold = 3

    # Create a new column to store data without outliers
    data[column_name+'_no_outliers'] = np.where(np.abs(z_scores) > threshold,␣
 ↪np.nan, data[column_name])

# Method 2: IQR Based Filtering
def iqr_outlier_treatment(data, column_name):
    # Calculate the first quartile (Q1) and third quartile (Q3)
    Q1 = data[column_name].quantile(0.25)
    Q3 = data[column_name].quantile(0.75)

    # Calculate the interquartile range (IQR)
    IQR = Q3 - Q1

    # Define upper and lower bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Create a new column to store data without outliers
    data[column_name+'_no_outliers'] = np.where((data[column_name] <␣
 ↪lower_bound) | (data[column_name] > upper_bound), np.nan, data[column_name])

# Method 3: Percentile Method (Winsorization)
def percentile_outlier_treatment(data, column_name):
    # Define the percentiles for lower and upper limits (e.g., 1% and 99%)
    lower_percentile = 1
    upper_percentile = 99

    # Calculate the lower and upper limits based on percentiles
    lower_limit = np.percentile(data[column_name], lower_percentile)
    upper_limit = np.percentile(data[column_name], upper_percentile)

    # Create a new column to store data without outliers
    data[column_name+'_no_outliers'] = np.where((data[column_name] <␣
 ↪lower_limit) | (data[column_name] > upper_limit), np.nan, data[column_name])

# Apply outlier treatment methods to the 'total_bill' column
column_name = 'total_bill'
```

```
z_score_outlier_treatment(data, column_name)
iqr_outlier_treatment(data, column_name)
percentile_outlier_treatment(data, column_name)

# Plot the results of outlier treatment
plot_with_outliers(data, column_name)
```

```
Original Data:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```



2. Implement the techniques to deal with missing values. https://note.nkmk.me/en/python-pandas-interpolate/ https://www.kdnuggets.com/2022/07/scikitlearn-imputer.html#:~:text=The%20imputer%20is%20an%20estimator,frequently%20used%20and%20constant%2 https://www.geeksforgeeks.org/principal-component-analysis-with-python/

[9]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target
```

```
# Introduce missing values artificially
missing_mask = np.random.rand(*X.shape) < 0.2  # 20% missing values
X_with_missing = X.copy()
X_with_missing[missing_mask] = np.nan

# Create a DataFrame for better visualization
iris_df = pd.DataFrame(data=np.column_stack((X_with_missing, y)), columns=iris.
 ↪feature_names + ['target'])

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_with_missing, y,␣
 ↪test_size=0.2, random_state=42)

# Impute missing values using SimpleImputer (mean strategy)
imputer = SimpleImputer(strategy='mean')
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)

# Train a RandomForestClassifier on the imputed data
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train_imputed, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test_imputed)

# Calculate accuracy on the test set
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy on the test set after imputation: {accuracy:.2f}")
```

```
Accuracy on the test set after imputation: 1.00
```

0.2 CO-2 ASSIGNMENT:

3. Implement distance measuring techniques for two features of your dataset: (a) Euclidean (b)Minkowski (c) Manhattan (d) Jaccard (e) Cosine (f) Simple matching coefficient (g)hamming (distance libraries-numpy, scipy, math)

[10]:
```
import numpy as np
from scipy.spatial import distance
import math
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
feature_names = iris.feature_names  # Feature names
```

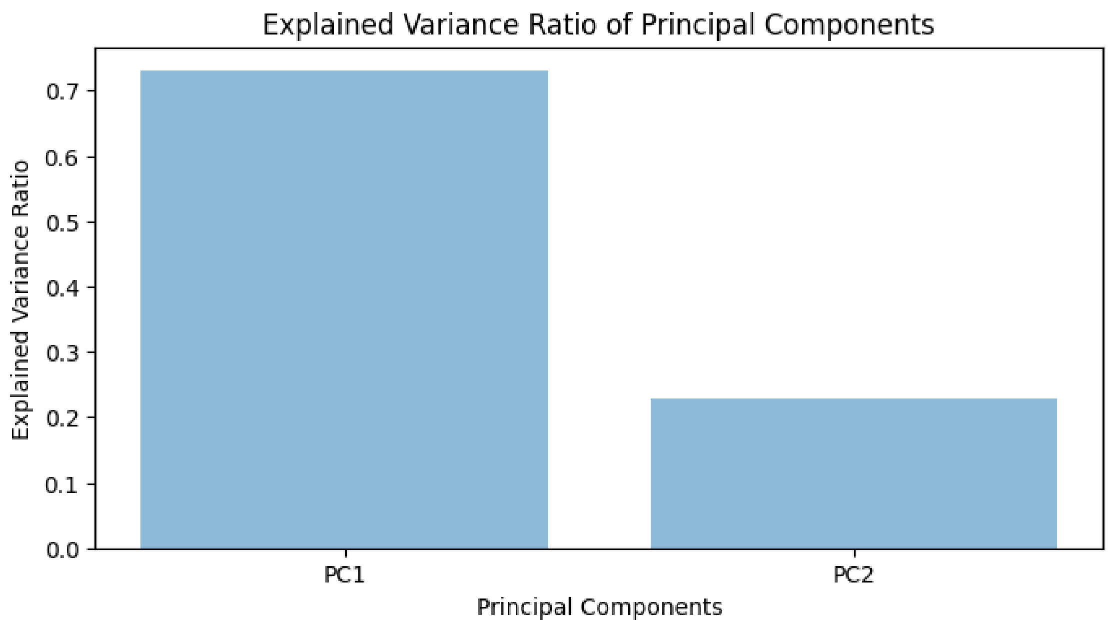
```python
# Select two features: Sepal Length (feature 0) and Sepal Width (feature 1)
feature1 = X[:, 0]
feature2 = X[:, 1]

# (a) Euclidean Distance
euclidean_dist = np.linalg.norm(feature1 - feature2)

# (b) Minkowski Distance (p=3 for example)
p = 3
minkowski_dist = distance.minkowski(feature1, feature2, p=p)

# (c) Manhattan Distance
manhattan_dist = distance.cityblock(feature1, feature2)

# (d) Jaccard Distance (for binary data, e.g., sets)
# Since the features are continuous, Jaccard distance is not applicable here

# (e) Cosine Similarity (use 1 - Cosine similarity for Cosine distance)
cosine_dist = 1 - np.dot(feature1, feature2) / (np.linalg.norm(feature1) * np.
 ↪linalg.norm(feature2))

# (f) Simple Matching Coefficient
# Since the features are continuous, SMC distance is not applicable here

# (g) Hamming Distance (for binary data, e.g., strings)
# Since the features are continuous, Hamming distance is not applicable here

# Print the calculated distances
print(f"(a) Euclidean Distance: {euclidean_dist:.2f}")
print(f"(b) Minkowski Distance (p={p}): {minkowski_dist:.2f}")
print(f"(c) Manhattan Distance: {manhattan_dist:.2f}")
print(f"(e) Cosine Distance: {cosine_dist:.2f}")
```

```
(a) Euclidean Distance: 36.16
(b) Minkowski Distance (p=3): 16.42
(c) Manhattan Distance: 417.90
(e) Cosine Distance: 0.02
```

4. Implement any data reduction technique.

[11]:
```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
```

```
X = iris.data  # Features
y = iris.target  # Target variable
feature_names = iris.feature_names

# Standardize the data (important for PCA)
mean = np.mean(X, axis=0)
std_dev = np.std(X, axis=0)
X_standardized = (X - mean) / std_dev

# Apply PCA to reduce dimensionality
n_components = 2  # Number of components to keep
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_standardized)

# Create a DataFrame with the reduced data
pca_df = pd.DataFrame(data=X_pca, columns=[f'PC{i+1}' for i in␣
 ↪range(n_components)])

# Concatenate the reduced data with the target variable
final_df = pd.concat([pca_df, pd.Series(y, name='target')], axis=1)

# Explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_

# Plot the explained variance ratio
plt.figure(figsize=(8, 4))
plt.bar(range(n_components), explained_variance_ratio, alpha=0.5,␣
 ↪align='center')
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.xticks(range(n_components), [f'PC{i+1}' for i in range(n_components)])
plt.title('Explained Variance Ratio of Principal Components')
plt.show()

# Display the first few rows of the reduced data
print(final_df.head())
```




```
        PC1       PC2  target
0 -2.264703  0.480027       0
1 -2.080961 -0.674134       0
2 -2.364229 -0.341908       0
3 -2.299384 -0.597395       0
4 -2.389842  0.646835       0
```

0.3 CO-3 ASSIGNMENT:

5. Implement various knn classification algorithms and do prediction for unknown data.

[12]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target variable

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

```python
# Define k-NN classifiers with different algorithms
knn_euclidean = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
knn_manhattan = KNeighborsClassifier(n_neighbors=3, metric='manhattan')
knn_chebyshev = KNeighborsClassifier(n_neighbors=3, metric='chebyshev')

# Train the k-NN classifiers on the training data
knn_euclidean.fit(X_train, y_train)
knn_manhattan.fit(X_train, y_train)
knn_chebyshev.fit(X_train, y_train)

# Predict the classes for the test data
y_pred_euclidean = knn_euclidean.predict(X_test)
y_pred_manhattan = knn_manhattan.predict(X_test)
y_pred_chebyshev = knn_chebyshev.predict(X_test)

# Calculate accuracy for each k-NN classifier
accuracy_euclidean = accuracy_score(y_test, y_pred_euclidean)
accuracy_manhattan = accuracy_score(y_test, y_pred_manhattan)
accuracy_chebyshev = accuracy_score(y_test, y_pred_chebyshev)

# Print the accuracy results
print("Accuracy (Euclidean Distance): {:.2f}".format(accuracy_euclidean))
print("Accuracy (Manhattan Distance): {:.2f}".format(accuracy_manhattan))
print("Accuracy (Chebyshev Distance): {:.2f}".format(accuracy_chebyshev))
```

```
Accuracy (Euclidean Distance): 1.00
Accuracy (Manhattan Distance): 1.00
Accuracy (Chebyshev Distance): 1.00
```

6. Implement a decision tree classification algorithm.

[13]:
```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target variable
feature_names = iris.feature_names

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

```
# Create a Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)

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

# Predict the classes for the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Generate a classification report
class_report = classification_report(y_test, y_pred, target_names=iris.
 ↪target_names)
print("Classification Report:\n", class_report)
```

```
Accuracy: 1.0
Classification Report:
               precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

7. Implement a support vector machine algorithm.

[14]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target variable

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

```
# Create a Support Vector Machine (SVM) classifier
clf = SVC(kernel='linear', C=1, random_state=42)

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

# Predict the classes for the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Generate a classification report
class_report = classification_report(y_test, y_pred, target_names=iris.
↪target_names)
print("Classification Report:\n", class_report)
```

```
Accuracy: 1.0
Classification Report:
               precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

8. Implement regression algorithms: (a)linear regression(b)logistic regression

[15]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load the California housing dataset
california_housing = fetch_california_housing()

# Use california_housing.data as features and california_housing.target as the␣
↪target variable
X = california_housing.data
y = california_housing.target
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Create a Linear Regression model
lr = LinearRegression()

# Train the model on the training data
lr.fit(X_train, y_train)

# Predict the target variable for the test data
y_pred = lr.predict(X_test)

# Calculate Mean Squared Error (MSE) and R-squared (R2) score
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", mse)
print("R-squared (R2) Score:", r2)
```

```
Mean Squared Error (MSE): 0.5558915986952422
R-squared (R2) Score: 0.5757877060324524
```

0.4 CO-4 ASSIGNMENT:

9. Implement k-means/k-medoid clustering algorithms and do prediction for unknown data.

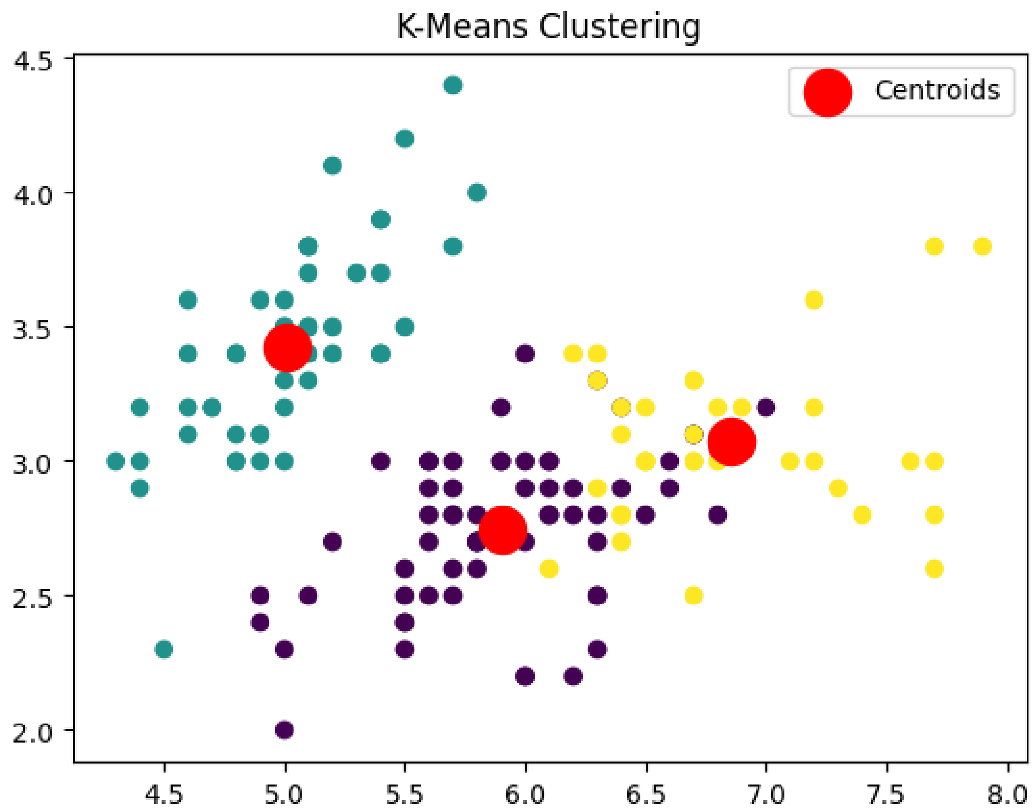
[16]:
```
pip install scikit-learn-extra
```

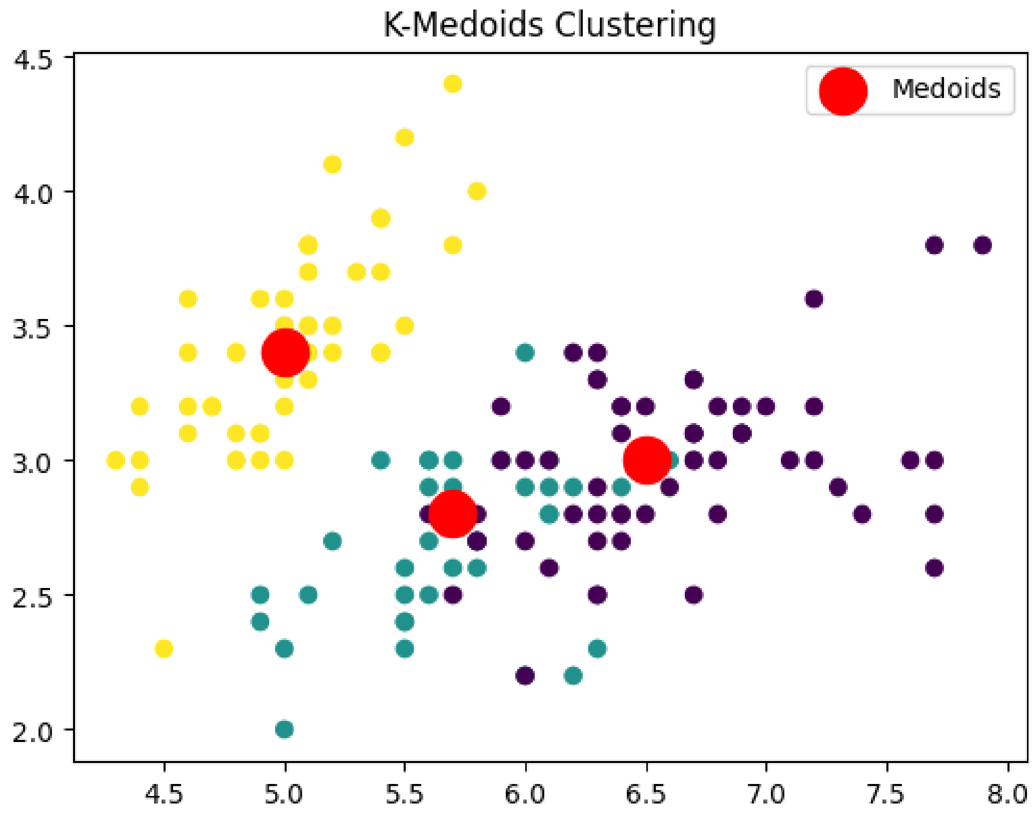
```
Collecting scikit-learn-extra
  Downloading scikit_learn_extra-0.3.0-cp310-cp310-manylinux_2_17_x86_64.manylin
ux2014_x86_64.whl (2.0 MB)
                                        2.0/2.0 MB
9.5 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.13.3 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn-extra) (1.23.5)
Requirement already satisfied: scipy>=0.19.1 in /usr/local/lib/python3.10/dist-
packages (from scikit-learn-extra) (1.11.3)
Requirement already satisfied: scikit-learn>=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn-extra) (1.2.2)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-
packages (from scikit-learn>=0.23.0->scikit-learn-extra) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.23.0->scikit-
learn-extra) (3.2.0)
Installing collected packages: scikit-learn-extra
Successfully installed scikit-learn-extra-0.3.0
```

[17]:

```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn_extra.cluster import KMedoids
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features

# Perform K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)

# Perform K-Medoids clustering
kmedoids = KMedoids(n_clusters=3, random_state=42)
kmedoids.fit(X)

# Predict clusters for the data points
kmeans_labels = kmeans.predict(X)
kmedoids_labels = kmedoids.predict(X)

# Visualize the clusters for K-Means
plt.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],␣
 ↪s=300, c='red', label='Centroids')
plt.title('K-Means Clustering')
plt.legend()
plt.show()

# Visualize the clusters for K-Medoids
plt.scatter(X[:, 0], X[:, 1], c=kmedoids_labels, cmap='viridis')
plt.scatter(kmedoids.cluster_centers_[:, 0], kmedoids.cluster_centers_[:, 1],␣
 ↪s=300, c='red', label='Medoids')
plt.title('K-Medoids Clustering')
plt.legend()
plt.show()

# Predict clusters for unknown data points
unknown_data = np.array([[5.1, 3.5, 1.4, 0.2], [6.5, 3.0, 5.2, 2.0]])  #␣
 ↪Replace with your own data
kmeans_prediction = kmeans.predict(unknown_data)
kmedoids_prediction = kmedoids.predict(unknown_data)

print("K-Means Prediction for Unknown Data:", kmeans_prediction)
print("K-Medoids Prediction for Unknown Data:", kmedoids_prediction)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870:
FutureWarning: The default value of `n_init` will change from 10 to 'auto' in
1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
```



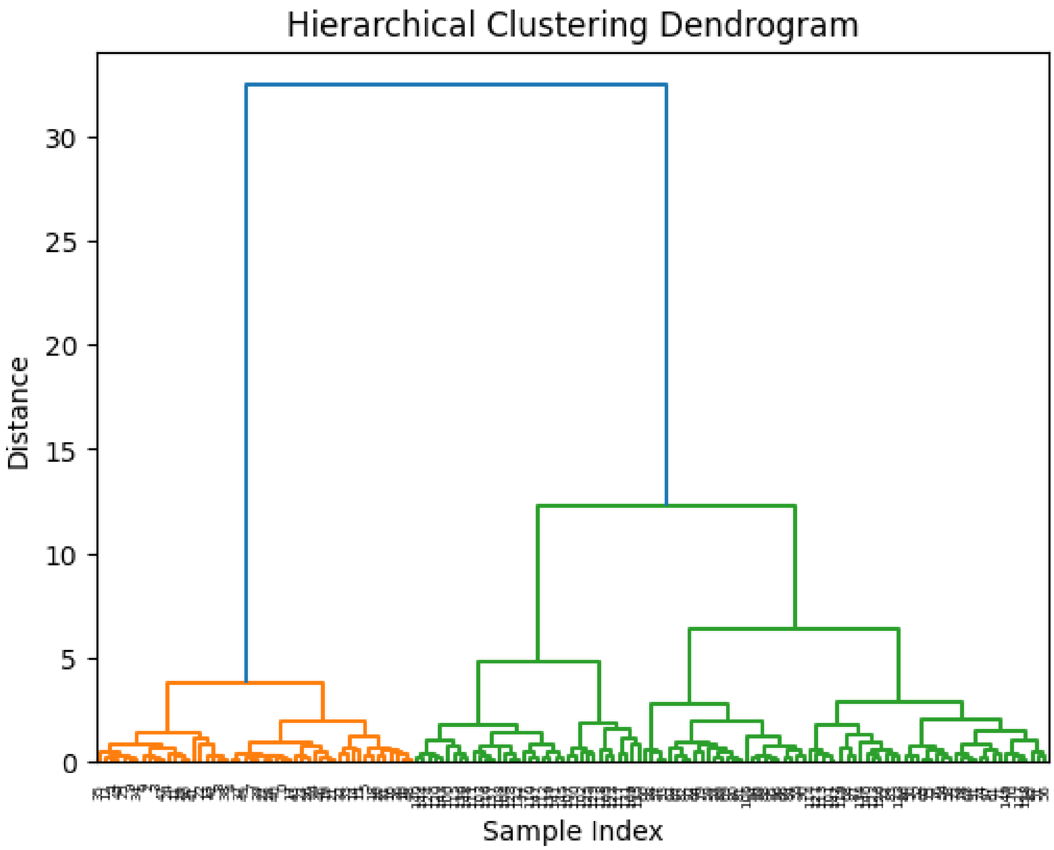



```
K-Means Prediction for Unknown Data: [1 2]
K-Medoids Prediction for Unknown Data: [2 0]
```

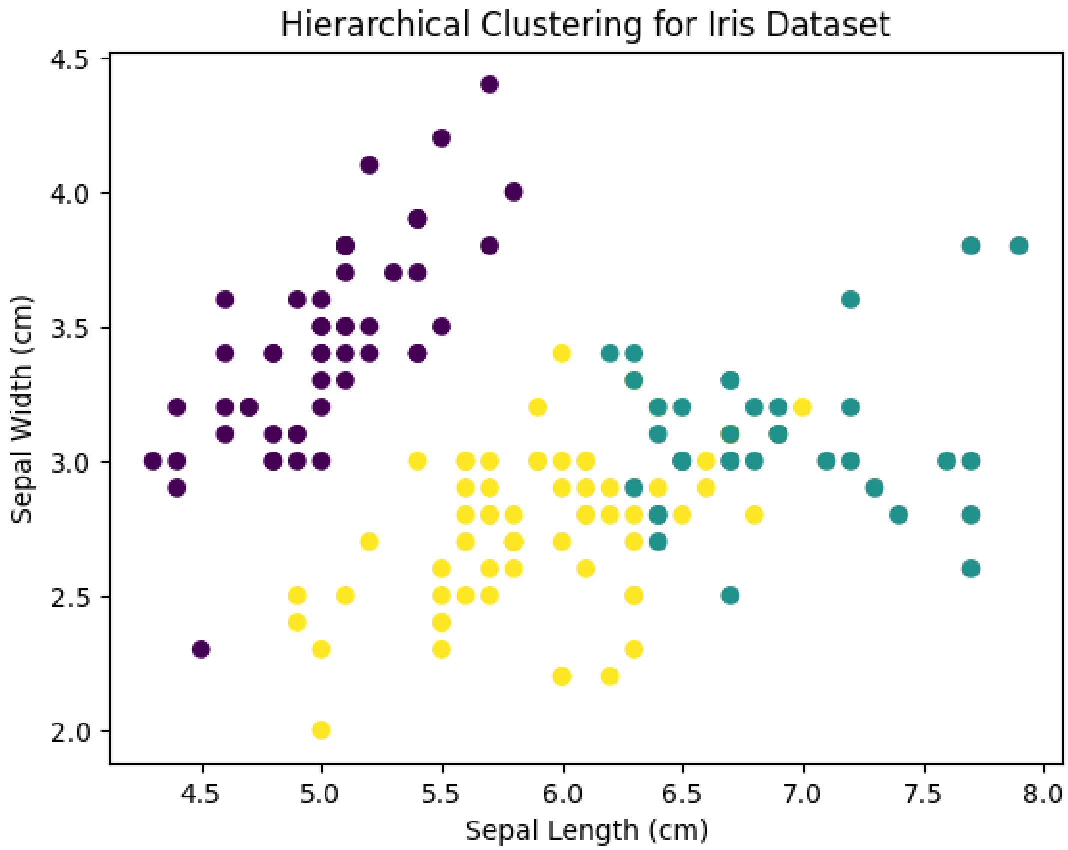
10. Implement hierarchical clustering algorithms and do prediction for unknown data.

[18]:
```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features

# Perform hierarchical clustering
linkage_matrix = linkage(X, method='ward', metric='euclidean')

# Create a dendrogram
dendrogram(linkage_matrix)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Sample Index')
```

```
plt.ylabel('Distance')
plt.show()

# Determine the number of clusters using the dendrogram
num_clusters = 3  # Adjust this based on the dendrogram

# Perform clustering to assign data points to clusters
clusters = fcluster(linkage_matrix, t=num_clusters, criterion='maxclust')

# Visualize the clusters for the Iris dataset
plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis')
plt.title('Hierarchical Clustering for Iris Dataset')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.show()

# Predict clusters for unknown data points
unknown_data = np.array([[5.1, 3.5, 1.4, 0.2], [6.5, 3.0, 5.2, 2.0]])  #␣
↪Replace with your own data

# Rebuild the linkage matrix with the unknown data points
linkage_matrix_unknown = linkage(unknown_data, method='ward',␣
↪metric='euclidean')

# Assign clusters to the unknown data points
unknown_clusters = fcluster(linkage_matrix, t=num_clusters,␣
↪criterion='maxclust')

print("Clusters for Unknown Data:", unknown_clusters)
```



Hierarchical Clustering Dendrogram
Distance
0
5
10
15
20
25
30
Sample Index

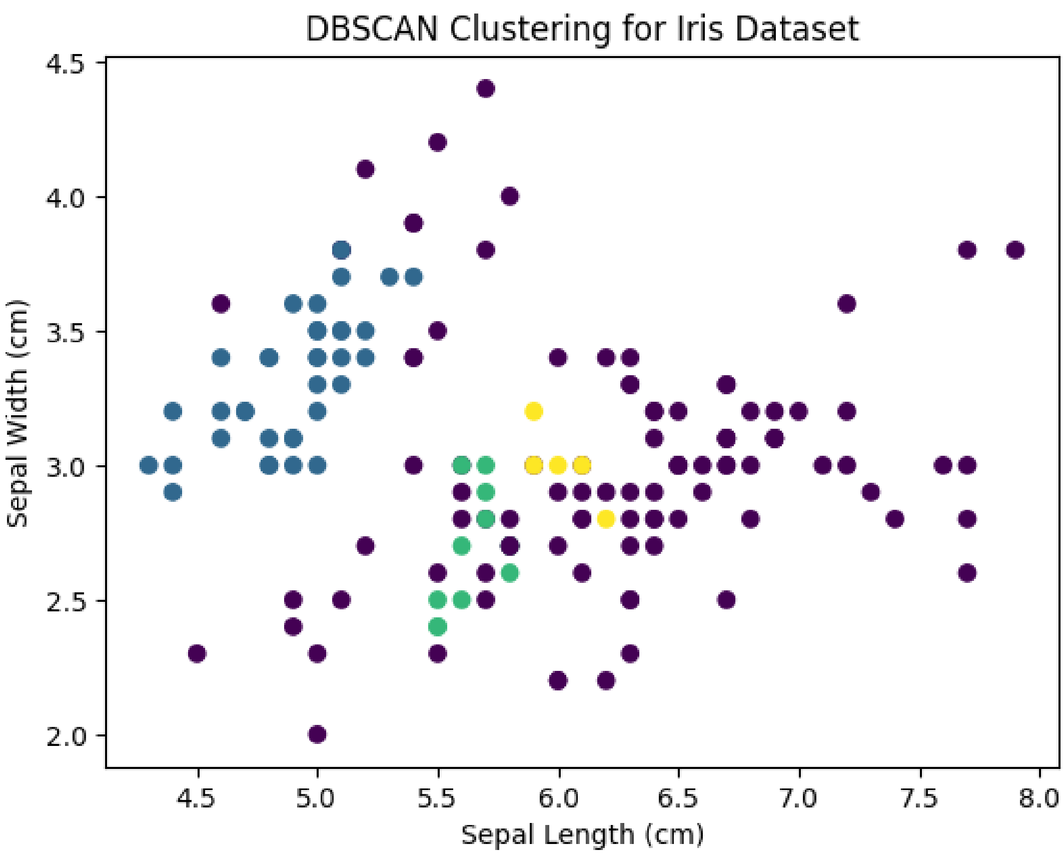



```
Clusters for Unknown Data: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 3 2 2 2 2 3 2 2 2 2
 2 2 3 3 2 2 2 2 3 2 3 2 3 2 2 3 3 2 2 2 2 2 3 3 2 2 2 3 2 2 2 3 2 2 2 3 2
 2 3]
```

11. Implement DBSCAN clustering algorithms and do prediction for unknown data.

```
[19]: import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features

# Perform DBSCAN clustering
dbscan = DBSCAN(eps=0.3, min_samples=5)
```

```
clusters = dbscan.fit_predict(X)

# Visualize the clusters
plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis')
plt.title('DBSCAN Clustering for Iris Dataset')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.show()

# Predict clusters for unknown data points (replace with your own data)
unknown_data = np.array([[5.1, 3.5, 1.4, 0.2], [6.5, 3.0, 5.2, 2.0]])  #␣
 ↪Replace with your own data
unknown_clusters = dbscan.fit_predict(unknown_data)

print("Clusters for Unknown Data:", unknown_clusters)
```




```
Clusters for Unknown Data: [-1 -1]
```

12. Implement apriori algorithm to get association rules.

[20]:
```python
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
import pandas as pd

# Sample transaction data (replace with your own dataset)
data = pd.DataFrame({
    'TransactionID': [1, 2, 3, 4, 5],
    'Items': ['A, B, D', 'B, C', 'A, C, D', 'A, D', 'B, C']
})

# Split items in the 'Items' column and create binary columns
items_df = data['Items'].str.get_dummies(', ')

# Concatenate the binary columns with the original DataFrame
data = pd.concat([data, items_df], axis=1)

# Drop the original 'Items' column
data.drop('Items', axis=1, inplace=True)

# Apply Apriori algorithm
frequent_itemsets = apriori(data.drop('TransactionID', axis=1), min_support=0.
↪5, use_colnames=True)

# Generate association rules
rules = association_rules(frequent_itemsets, metric='lift', min_threshold=1.0)

# Display association rules
print("Association Rules:")
print(rules)
```

```
Association Rules:
  antecedents consequents  antecedent support  consequent support  support  \
0         (D)         (A)                 0.6                 0.6      0.6
1         (A)         (D)                 0.6                 0.6      0.6

   confidence      lift  leverage  conviction  zhangs_metric
0         1.0  1.666667      0.24         inf            1.0
1         1.0  1.666667      0.24         inf            1.0
```

```
/usr/local/lib/python3.10/dist-
packages/mlxtend/frequent_patterns/fpcommon.py:110: DeprecationWarning:
DataFrames with non-bool types result in worse computationalperformance and
their support might be discontinued in the future.Please use a DataFrame with
bool type
  warnings.warn(
```

13. Implement backpropagation neural network algorithm.

[21]:
```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target variable

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Create and train the neural network
clf = MLPClassifier(hidden_layer_sizes=(10, 5), max_iter=1000, random_state=42)
clf.fit(X_train, y_train)

# Predict the target variable
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to `transformed_cell`
argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
Accuracy: 0.9666666666666667
```

```
/usr/local/lib/python3.10/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:686:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (1000) reached and
the optimization hasn't converged yet.
  warnings.warn(
```

14. Make a comparison tables for classification and clustering algorithms, for what you implemented here:

(a) Write unknown data:

(b)Compare performance of classification algorithms

[22]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score

# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)

# Train the models
knn_model = KNeighborsClassifier()
knn_model.fit(X_train, y_train)
y_pred_knn = knn_model.predict(X_test)

dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict(X_test)

svm_model = SVC()
svm_model.fit(X_train, y_train)
y_pred_svm = svm_model.predict(X_test)

# Calculate evaluation metrics
accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn, average='micro')
recall_knn = recall_score(y_test, y_pred_knn, average='micro')
f1_knn = f1_score(y_test, y_pred_knn, average='micro')

accuracy_dt = accuracy_score(y_test, y_pred_dt)
precision_dt = precision_score(y_test, y_pred_dt, average='micro')
recall_dt = recall_score(y_test, y_pred_dt, average='micro')
f1_dt = f1_score(y_test, y_pred_dt, average='micro')

accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm, average='micro')
recall_svm = recall_score(y_test, y_pred_svm, average='micro')
f1_svm = f1_score(y_test, y_pred_svm, average='micro')

# Printing results in a table
print("Comparison Table for Classification Algorithms:")
print("{:<15} {:<10} {:<12} {:<10} {:<10} {:<10} {:<20}".format('Algorithm␣
↪name', 'Accuracy', 'Sensitivity', 'F-measure', 'Precision', 'Recall',␣
↪'Prediction for unknown data'))
```

```
print("{:<15} {:<10.2f} {:<12.2f} {:<10.2f} {:<10.2f} {:<10.2f} {:<20}".
 ↪format('KNN', accuracy_knn, recall_knn, f1_knn, precision_knn, recall_knn,␣
 ↪str(y_pred_knn)))
print("{:<15} {:<10.2f} {:<12.2f} {:<10.2f} {:<10.2f} {:<10.2f} {:<20}".
 ↪format('Decision Tree', accuracy_dt, recall_dt, f1_dt, precision_dt,␣
 ↪recall_dt, str(y_pred_dt)))
print("{:<15} {:<10.2f} {:<12.2f} {:<10.2f} {:<10.2f} {:<10.2f} {:<20}".
 ↪format('SVM', accuracy_svm, recall_svm, f1_svm, precision_svm, recall_svm,␣
 ↪str(y_pred_svm)))
```

```
Comparison Table for Classification Algorithms:
Algorithm name  Accuracy   Sensitivity  F-measure  Precision  Recall
Prediction for unknown data
KNN             1.00       1.00         1.00       1.00       1.00       [1 0 2
1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0]
Decision Tree   1.00       1.00         1.00       1.00       1.00       [1 0 2
1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0]
SVM             1.00       1.00         1.00       1.00       1.00       [1 0 2
1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0]
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to `transformed_cell`
argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

(c) Compare performance of clustering algorithms you implemented.

[23]:
```
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_score

# Assuming you have your data stored in X

# K-means clustering
kmeans = KMeans(n_clusters=3, random_state=0)
kmeans.fit(X)
kmeans_labels = kmeans.labels_
kmeans_silhouette_score = silhouette_score(X, kmeans_labels)

# Agglomerative clustering
agg = AgglomerativeClustering(n_clusters=3)
agg.fit(X)
agg_labels = agg.labels_
agg_silhouette_score = silhouette_score(X, agg_labels)
```

```
# Printing the results
print("Comparison of Clustering Algorithms:")
print(f"K-means Silhouette Score: {kmeans_silhouette_score}")
print(f"Agglomerative Clustering Silhouette Score: {agg_silhouette_score}")
```

```
Comparison of Clustering Algorithms:
K-means Silhouette Score: 0.5528190123564095
Agglomerative Clustering Silhouette Score: 0.5543236611296419
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to `transformed_cell`
argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870:
FutureWarning: The default value of `n_init` will change from 10 to 'auto' in
1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
```

(d) Use different distance measures as in CO2's 3rd assignment and make a table to compare the performance of clustering algorithms you implemented.

[24]:
```
import numpy as np
from scipy.spatial.distance import cdist
from scipy.spatial.distance import cityblock, cosine, hamming

# Assuming you have already initialized X and the clustering algorithms

# Calculate distances for K-means
kmeans_distances = {
    'Euclidean': cdist(X, kmeans.cluster_centers_, 'euclidean'),
    'Minkowski': cdist(X, kmeans.cluster_centers_, 'minkowski', p=3),
    'Manhattan': cdist(X, kmeans.cluster_centers_, 'cityblock'),
    'Jaccard': cdist(X, kmeans.cluster_centers_, 'jaccard'),
    'Cosine': cdist(X, kmeans.cluster_centers_, 'cosine'),
    'Simple matching coefficient': cdist(X, kmeans.cluster_centers_, 'hamming')
}

# Calculate distances for Agglomerative clustering
agg_distances = {
    'Euclidean': cdist(X, np.array([np.mean(X, axis=0)]), 'euclidean'),
    'Minkowski': cdist(X, np.array([np.mean(X, axis=0)]), 'minkowski', p=3),
    'Manhattan': cdist(X, np.array([np.mean(X, axis=0)]), 'cityblock'),
    'Jaccard': cdist(X, np.array([np.mean(X, axis=0)]), 'jaccard'),
    'Cosine': cdist(X, np.array([np.mean(X, axis=0)]), 'cosine'),
```

```
    'Simple matching coefficient': cdist(X, np.array([np.mean(X, axis=0)]),␣
↪'hamming')
}

# Create a table to compare the performance of clustering algorithms using␣
↪different distance measures
print("Comparison Table for Clustering Algorithms with Different Distance␣
↪Measures:")
print("{:<30} {:<15} {:<15}".format('Distance Measure', 'K-means',␣
↪'Agglomerative'))
for key in kmeans_distances:
    print("{:<30} {:<15} {:<15}".format(key, np.mean(kmeans_distances[key]), np.
↪mean(agg_distances[key])))

kmeans_avg_distance = np.mean([np.mean(kmeans_distances[key]) for key in␣
↪kmeans_distances])
agg_avg_distance = np.mean([np.mean(agg_distances[key]) for key in␣
↪agg_distances])

if kmeans_avg_distance < agg_avg_distance:
    print("K-means clustering is better for this data based on average distance.
↪")
elif kmeans_avg_distance > agg_avg_distance:
    print("Agglomerative clustering is better for this data based on average␣
↪distance.")
else:
    print("Both clustering algorithms perform equally well on this data based␣
↪on average distance.")
```

```
Comparison Table for Clustering Algorithms with Different Distance Measures:
Distance Measure               K-means         Agglomerative
Euclidean                      2.4640149137174205 1.9440683605553901
Minkowski                      2.191052127529947 1.7293217917093848
Manhattan                      4.149625097151481 3.2452177777777766
Jaccard                        1.0             1.0
Cosine                         0.04404597214669159 0.02301730036009452
Simple matching coefficient    1.0             1.0
Agglomerative clustering is better for this data based on average distance.
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to `transformed_cell`
argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

15. Write any deep learning program of your choice.

[25]:
```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target

# Preprocess the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Build the neural network
model = Sequential()
model.add(Dense(10, input_dim=4, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(3, activation='softmax'))  # 3 classes for the Iris dataset

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',␣
↪metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=150, batch_size=10, verbose=1)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f'Test loss: {loss:.4f}')
print(f'Test accuracy: {accuracy:.4f}')
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to `transformed_cell`
argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
Epoch 1/150
12/12 [==============================] - 1s 2ms/step - loss: 1.0917 - accuracy:
0.3500
Epoch 2/150
12/12 [==============================] - 0s 2ms/step - loss: 1.0633 - accuracy:
0.4333
```

```
Epoch 3/150
12/12 [==============================] - 0s 2ms/step - loss: 1.0370 - accuracy:
0.6333
Epoch 4/150
12/12 [==============================] - 0s 2ms/step - loss: 1.0153 - accuracy:
0.6583
Epoch 5/150
12/12 [==============================] - 0s 2ms/step - loss: 0.9935 - accuracy:
0.6583
Epoch 6/150
12/12 [==============================] - 0s 2ms/step - loss: 0.9694 - accuracy:
0.6667
Epoch 7/150
12/12 [==============================] - 0s 2ms/step - loss: 0.9453 - accuracy:
0.6667
Epoch 8/150
12/12 [==============================] - 0s 2ms/step - loss: 0.9199 - accuracy:
0.6667
Epoch 9/150
12/12 [==============================] - 0s 2ms/step - loss: 0.8939 - accuracy:
0.6667
Epoch 10/150
12/12 [==============================] - 0s 3ms/step - loss: 0.8669 - accuracy:
0.6750
Epoch 11/150
12/12 [==============================] - 0s 2ms/step - loss: 0.8411 - accuracy:
0.6750
Epoch 12/150
12/12 [==============================] - 0s 2ms/step - loss: 0.8149 - accuracy:
0.6667
Epoch 13/150
12/12 [==============================] - 0s 2ms/step - loss: 0.7915 - accuracy:
0.6750
Epoch 14/150
12/12 [==============================] - 0s 2ms/step - loss: 0.7688 - accuracy:
0.6750
Epoch 15/150
12/12 [==============================] - 0s 3ms/step - loss: 0.7497 - accuracy:
0.6750
Epoch 16/150
12/12 [==============================] - 0s 2ms/step - loss: 0.7314 - accuracy:
0.6750
Epoch 17/150
12/12 [==============================] - 0s 2ms/step - loss: 0.7146 - accuracy:
0.6750
Epoch 18/150
12/12 [==============================] - 0s 2ms/step - loss: 0.6991 - accuracy:
0.6750
```

```
Epoch 19/150
12/12 [==============================] - 0s 2ms/step - loss: 0.6845 - accuracy:
0.6750
Epoch 20/150
12/12 [==============================] - 0s 2ms/step - loss: 0.6694 - accuracy:
0.6750
Epoch 21/150
12/12 [==============================] - 0s 4ms/step - loss: 0.6571 - accuracy:
0.6667
Epoch 22/150
12/12 [==============================] - 0s 3ms/step - loss: 0.6441 - accuracy:
0.6667
Epoch 23/150
12/12 [==============================] - 0s 3ms/step - loss: 0.6320 - accuracy:
0.6667
Epoch 24/150
12/12 [==============================] - 0s 3ms/step - loss: 0.6211 - accuracy:
0.6583
Epoch 25/150
12/12 [==============================] - 0s 3ms/step - loss: 0.6103 - accuracy:
0.6583
Epoch 26/150
12/12 [==============================] - 0s 4ms/step - loss: 0.6007 - accuracy:
0.6583
Epoch 27/150
12/12 [==============================] - 0s 3ms/step - loss: 0.5917 - accuracy:
0.6583
Epoch 28/150
12/12 [==============================] - 0s 3ms/step - loss: 0.5819 - accuracy:
0.6333
Epoch 29/150
12/12 [==============================] - 0s 4ms/step - loss: 0.5726 - accuracy:
0.6417
Epoch 30/150
12/12 [==============================] - 0s 4ms/step - loss: 0.5647 - accuracy:
0.6583
Epoch 31/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5579 - accuracy:
0.6333
Epoch 32/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5508 - accuracy:
0.5917
Epoch 33/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5448 - accuracy:
0.5583
Epoch 34/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5389 - accuracy:
0.5667
```

```
Epoch 35/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5332 - accuracy:
0.4667
Epoch 36/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5283 - accuracy:
0.6083
Epoch 37/150
12/12 [==============================] - 0s 3ms/step - loss: 0.5237 - accuracy:
0.6583
Epoch 38/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5198 - accuracy:
0.6583
Epoch 39/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5155 - accuracy:
0.6583
Epoch 40/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5122 - accuracy:
0.6583
Epoch 41/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5082 - accuracy:
0.6583
Epoch 42/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5047 - accuracy:
0.6583
Epoch 43/150
12/12 [==============================] - 0s 2ms/step - loss: 0.5012 - accuracy:
0.6583
Epoch 44/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4982 - accuracy:
0.6583
Epoch 45/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4951 - accuracy:
0.6583
Epoch 46/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4910 - accuracy:
0.6583
Epoch 47/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4866 - accuracy:
0.6833
Epoch 48/150
12/12 [==============================] - 0s 3ms/step - loss: 0.4818 - accuracy:
0.7417
Epoch 49/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4777 - accuracy:
0.8167
Epoch 50/150
12/12 [==============================] - 0s 3ms/step - loss: 0.4721 - accuracy:
0.8333
```

```
Epoch 51/150
12/12 [==============================] - 0s 3ms/step - loss: 0.4668 - accuracy:
0.8583
Epoch 52/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4611 - accuracy:
0.9250
Epoch 53/150
12/12 [==============================] - 0s 3ms/step - loss: 0.4549 - accuracy:
0.9250
Epoch 54/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4485 - accuracy:
0.9250
Epoch 55/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4430 - accuracy:
0.9167
Epoch 56/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4361 - accuracy:
0.9333
Epoch 57/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4309 - accuracy:
0.9167
Epoch 58/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4235 - accuracy:
0.9250
Epoch 59/150
12/12 [==============================] - 0s 3ms/step - loss: 0.4169 - accuracy:
0.9417
Epoch 60/150
12/12 [==============================] - 0s 3ms/step - loss: 0.4110 - accuracy:
0.9333
Epoch 61/150
12/12 [==============================] - 0s 2ms/step - loss: 0.4063 - accuracy:
0.9250
Epoch 62/150
12/12 [==============================] - 0s 2ms/step - loss: 0.3980 - accuracy:
0.9333
Epoch 63/150
12/12 [==============================] - 0s 3ms/step - loss: 0.3915 - accuracy:
0.9500
Epoch 64/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3863 - accuracy:
0.9583
Epoch 65/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3789 - accuracy:
0.9500
Epoch 66/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3718 - accuracy:
0.9583
```

```
Epoch 67/150
12/12 [==============================] - 0s 3ms/step - loss: 0.3687 - accuracy:
0.9500
Epoch 68/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3597 - accuracy:
0.9583
Epoch 69/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3536 - accuracy:
0.9583
Epoch 70/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3499 - accuracy:
0.9417
Epoch 71/150
12/12 [==============================] - 0s 5ms/step - loss: 0.3445 - accuracy:
0.9417
Epoch 72/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3343 - accuracy:
0.9667
Epoch 73/150
12/12 [==============================] - 0s 5ms/step - loss: 0.3292 - accuracy:
0.9583
Epoch 74/150
12/12 [==============================] - 0s 4ms/step - loss: 0.3216 - accuracy:
0.9583
Epoch 75/150
12/12 [==============================] - 0s 3ms/step - loss: 0.3183 - accuracy:
0.9583
Epoch 76/150
12/12 [==============================] - 0s 3ms/step - loss: 0.3101 - accuracy:
0.9583
Epoch 77/150
12/12 [==============================] - 0s 3ms/step - loss: 0.3039 - accuracy:
0.9667
Epoch 78/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2982 - accuracy:
0.9667
Epoch 79/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2925 - accuracy:
0.9583
Epoch 80/150
12/12 [==============================] - 0s 4ms/step - loss: 0.2870 - accuracy:
0.9667
Epoch 81/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2818 - accuracy:
0.9667
Epoch 82/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2772 - accuracy:
0.9583
```

```
Epoch 83/150
12/12 [==============================] - 0s 4ms/step - loss: 0.2760 - accuracy:
0.9667
Epoch 84/150
12/12 [==============================] - 0s 5ms/step - loss: 0.2682 - accuracy:
0.9583
Epoch 85/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2594 - accuracy:
0.9750
Epoch 86/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2569 - accuracy:
0.9667
Epoch 87/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2548 - accuracy:
0.9667
Epoch 88/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2462 - accuracy:
0.9667
Epoch 89/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2424 - accuracy:
0.9667
Epoch 90/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2377 - accuracy:
0.9750
Epoch 91/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2346 - accuracy:
0.9583
Epoch 92/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2299 - accuracy:
0.9667
Epoch 93/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2236 - accuracy:
0.9667
Epoch 94/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2219 - accuracy:
0.9833
Epoch 95/150
12/12 [==============================] - 0s 3ms/step - loss: 0.2181 - accuracy:
0.9667
Epoch 96/150
12/12 [==============================] - 0s 6ms/step - loss: 0.2151 - accuracy:
0.9667
Epoch 97/150
12/12 [==============================] - 0s 7ms/step - loss: 0.2075 - accuracy:
0.9750
Epoch 98/150
12/12 [==============================] - 0s 8ms/step - loss: 0.2085 - accuracy:
0.9667
```

```
Epoch 99/150
12/12 [==============================] - 0s 6ms/step - loss: 0.2024 - accuracy:
0.9667
Epoch 100/150
12/12 [==============================] - 0s 8ms/step - loss: 0.1995 - accuracy:
0.9667
Epoch 101/150
12/12 [==============================] - 0s 8ms/step - loss: 0.1972 - accuracy:
0.9750
Epoch 102/150
12/12 [==============================] - 0s 9ms/step - loss: 0.1918 - accuracy:
0.9750
Epoch 103/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1891 - accuracy:
0.9750
Epoch 104/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1887 - accuracy:
0.9750
Epoch 105/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1827 - accuracy:
0.9750
Epoch 106/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1806 - accuracy:
0.9667
Epoch 107/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1769 - accuracy:
0.9667
Epoch 108/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1763 - accuracy:
0.9667
Epoch 109/150
12/12 [==============================] - 0s 5ms/step - loss: 0.1739 - accuracy:
0.9833
Epoch 110/150
12/12 [==============================] - 0s 5ms/step - loss: 0.1729 - accuracy:
0.9667
Epoch 111/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1672 - accuracy:
0.9667
Epoch 112/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1658 - accuracy:
0.9750
Epoch 113/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1619 - accuracy:
0.9750
Epoch 114/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1614 - accuracy:
0.9750
```

```
Epoch 115/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1587 - accuracy:
0.9667
Epoch 116/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1567 - accuracy:
0.9750
Epoch 117/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1595 - accuracy:
0.9750
Epoch 118/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1517 - accuracy:
0.9667
Epoch 119/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1507 - accuracy:
0.9667
Epoch 120/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1492 - accuracy:
0.9750
Epoch 121/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1482 - accuracy:
0.9750
Epoch 122/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1451 - accuracy:
0.9750
Epoch 123/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1443 - accuracy:
0.9750
Epoch 124/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1411 - accuracy:
0.9750
Epoch 125/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1406 - accuracy:
0.9750
Epoch 126/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1391 - accuracy:
0.9750
Epoch 127/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1381 - accuracy:
0.9750
Epoch 128/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1350 - accuracy:
0.9667
Epoch 129/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1342 - accuracy:
0.9750
Epoch 130/150
12/12 [==============================] - 0s 6ms/step - loss: 0.1351 - accuracy:
0.9750
```

```
Epoch 131/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1308 - accuracy:
0.9750
Epoch 132/150
12/12 [==============================] - 0s 5ms/step - loss: 0.1303 - accuracy:
0.9750
Epoch 133/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1282 - accuracy:
0.9750
Epoch 134/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1282 - accuracy:
0.9750
Epoch 135/150
12/12 [==============================] - 0s 4ms/step - loss: 0.1262 - accuracy:
0.9750
Epoch 136/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1284 - accuracy:
0.9667
Epoch 137/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1241 - accuracy:
0.9750
Epoch 138/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1226 - accuracy:
0.9750
Epoch 139/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1223 - accuracy:
0.9750
Epoch 140/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1233 - accuracy:
0.9667
Epoch 141/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1207 - accuracy:
0.9750
Epoch 142/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1196 - accuracy:
0.9750
Epoch 143/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1177 - accuracy:
0.9750
Epoch 144/150
12/12 [==============================] - 0s 3ms/step - loss: 0.1229 - accuracy:
0.9750
Epoch 145/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1155 - accuracy:
0.9750
Epoch 146/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1156 - accuracy:
0.9750
```

```
Epoch 147/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1150 - accuracy:
0.9750
Epoch 148/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1135 - accuracy:
0.9750
Epoch 149/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1135 - accuracy:
0.9750
Epoch 150/150
12/12 [==============================] - 0s 2ms/step - loss: 0.1148 - accuracy:
0.9750
Test loss: 0.1701
Test accuracy: 0.8667
```