

VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEER



Lab 4 Report

Student: Le Thanh Hai - 10421016

Student: Trinh The Hao - 10421017

Student: Vu Ngoc Quang - 10421103

1 Exercise 2.1 & 2.2 & 2.3

```
1 from sklearn.datasets import load_iris
2 from sklearn.cluster import KMeans
3 from sklearn.metrics import accuracy_score
4
5 # Load dataset
6 iris = load_iris()
7
8 # divide dataset into features and labels
9 X = iris.data
10 y = iris.target
11
12 # create model
13 model = KMeans(n_clusters=3, random_state=0, n_init="auto").fit(X)
14
15 # predict output
16 predictions = model.predict(X)
17
18 # print accuracy
19 print("Accuracy is: ", round(accuracy_score(y, predictions)*100,2), "%"↵
    )
```

Explanation:

It performs K-means clustering on the iris dataset and prints the resulting accuracy of the clustering.

It first loads the iris dataset using the loadiris function from the sklearn.datasets module. It then divides the dataset into features and labels, where the features are the four measurements of each sample, and the labels are the correct classifications of each sample (setosa, versicolor, or virginica).

It creates a KMeans object with three clusters and fits it to the dataset using the fit method. The ninit="auto" parameter specifies that the algorithm should run multiple times with different initializations and choose the best result.

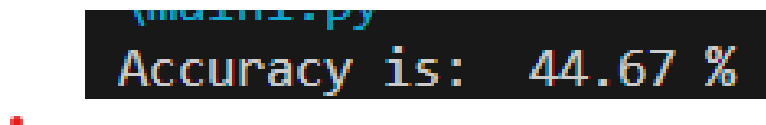


Figure 1: Kmean Accuracy

2 Exercise 2.4 & 2.5 & 2.6

```
1 from sklearn.model_selection import train_test_split
2 from sklearn.cluster import KMeans
3 from sklearn.metrics import accuracy_score
4 from sklearn.datasets import load_iris
5 import matplotlib.pyplot as plt
6
7 # Load dataset
8 iris = load_iris()
9
10 # divide dataset into features and labels
11 X = iris.data
12 y = iris.target
13
14 scores = []
15 wss = []
16 ks = range(2, 11)
17
18 for k in ks:
19     kmeans = KMeans(n_clusters=k, random_state=40, n_init="auto").fit(X)
20
21     pred = kmeans.predict(X)
22     score = round(accuracy_score(y, pred), 4)
23     scores.append(score)
24     wss.append(kmeans.inertia_)
25
26 plt.plot(ks, scores, color='purple')
27 plt.xlabel('Number of clusters')
28 plt.ylabel('Accuracy')
29 plt.title('Accuracy vs Number of clusters')
30 plt.show()
31
32 plt.plot(ks, wss, color='red')
33 plt.xlabel('k')
34 plt.ylabel('Within-Cluster-Sum of Squared Errors')
35 plt.title('Within-Cluster-Sum of Squared Errors with different k')
36 plt.show()
```

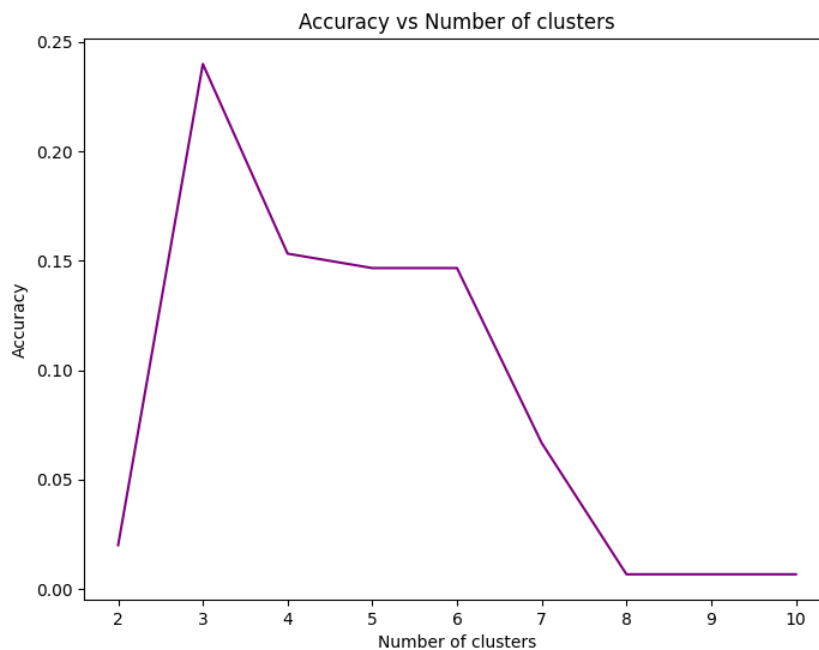
Explanation:

This code performs K-means clustering on the iris dataset and plots two graphs to help evaluate the clustering results.

It initializes two empty lists scores and wss, which will be used to store the accuracy and within-cluster-sum of squared errors (WSS) for each value of k. The range of k values to test is set to be from 2 to 10.

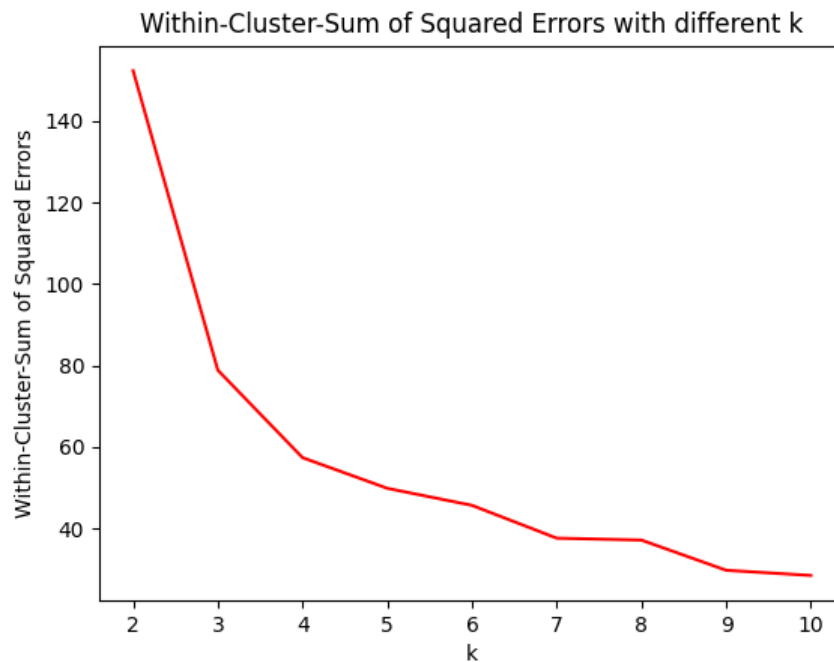
It iterates over each value of k and performs K-means clustering on the dataset using the `KMeans` class from the `sklearn.cluster` module. The `nclusters` parameter is set to the current value of k , and the `ninit="auto"` parameter specifies that the algorithm should run multiple times with different initializations and choose the best result.

It plots two graphs using the `matplotlib.pyplot` module. The first graph shows the accuracy of the clustering as a function of the number of clusters, with k on the x-axis and accuracy on the y-axis. The second graph shows the WSS as a function of k , with k on the x-axis and WSS on the y-axis. These graphs can be used to help determine the optimal number of clusters for the dataset.



3 Exercise 2.7 & 2.8

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.preprocessing import StandardScaler
4 from numpy.random import uniform
5 from sklearn.datasets import make_blobs
6 import seaborn as sns
7 import random
8 from sklearn.datasets import load_iris
9 from sklearn.model_selection import train_test_split
10 from sklearn.metrics import accuracy_score
11
12 def euclidean(point, data):
```



```

13     """
14     Euclidean distance between point & data.
15     Point has dimensions (m,), data has dimensions (n,m), and output ←
16         will be of size (n,).
17     """
18     return np.sqrt(np.sum((point - data)**2, axis=1))
19
20 class KMeans:
21     def __init__(self, n_clusters=8, max_iter=300):
22         self.n_clusters = n_clusters
23         self.max_iter = max_iter
24     def fit(self, X):
25         # Initialize the centroids, using the "k-means++" method, ←
26         # where a random datapoint is selected as the first,
27         # then the rest are initialized w/ probabilities proportional ←
28         # to their distances to the first
29         # Pick a random point from train data for first centroid
30         self.centroids = [random.choice(X)]
31
32         for _ in range(self.n_clusters-1):
33             # Calculate distances from points to the centroids
34             dists = np.sum([euclidean(centroid, X) for centroid in ←
35                             self.centroids], axis=0)
36             # Normalize the distances
37             dists /= np.sum(dists)
38             # Choose remaining points based on their distances

```

```
35         new_centroid_idx, = np.random.choice(range(len(X)), size=
           =1, p=dists)
36         self.centroids += [X[new_centroid_idx]]
37         # This initial method of randomly selecting centroid starts is
           less effective
38         # min_, max_ = np.min(X, axis=0), np.max(X, axis=0)
39         # self.centroids = [uniform(min_, max_) for _ in range(self.
           n_clusters)]
40         # Iterate, adjusting centroids until converged or until passed
           max_iter
41         iteration = 0
42         prev_centroids = None
43         while np.not_equal(self.centroids, prev_centroids).any() and
           iteration < self.max_iter:
44             # Sort each datapoint, assigning to nearest centroid
45             sorted_points = [[] for _ in range(self.n_clusters)]
46             for x in X:
47                 dists = euclidean(x, self.centroids)
48                 centroid_idx = np.argmin(dists)
49                 sorted_points[centroid_idx].append(x)
50             # Push current centroids to previous, reassign centroids
           as mean of the points belonging to them
51             prev_centroids = self.centroids
52             self.centroids = [np.mean(cluster, axis=0) for cluster in
           sorted_points]
53             for i, centroid in enumerate(self.centroids):
54                 if np.isnan(centroid).any(): # Catch any np.nans,
           resulting from a centroid having no points
55                     self.centroids[i] = prev_centroids[i]
56             iteration += 1
57
58     def evaluate(self, X):
59         centroids = []
60         centroid_idxxs = []
61         for x in X:
62             dists = euclidean(x, self.centroids)
63             centroid_idx = np.argmin(dists)
64             centroids.append(self.centroids[centroid_idx])
65             centroid_idxxs.append(centroid_idx)
66         return centroids, centroid_idxxs
67
68     def calculate_wcss(self, X):
69         wcss = 0
70         for i in range(self.n_clusters):
```

```
71         cluster_points = X[np.where(np.array(classification) == i)↵
72             ]
73         centroid = self.centroids[i]
74         cluster_error = np.sum((cluster_points - centroid) ** 2)
75         wcss += cluster_error
76     ↵
77     # Load dataset
78     iris = load_iris()
79
80     # divide dataset into features and labels
81     X = iris.data
82     y = iris.target
83
84
85     centers = 3
86     kmeans = KMeans(n_clusters=centers)
87     kmeans.fit(X)
88
89     # View results
90     true_labels = y
91     class_centers, classification = kmeans.evaluate(X)
92
93     # print accuracy
94     print("Accuracy is: ", round(accuracy_score(y, classification)*100,2),↵
95         "%")
96
97     # calculate wcss
98     wcss = kmeans.calculate_wcss(X)
99     print("WCSS is: ", wcss)
```

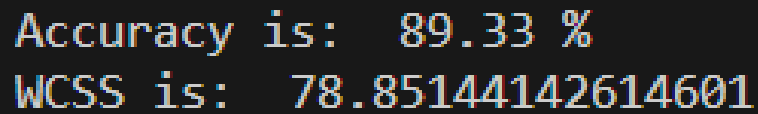
Explanation:

It defines a custom implementation of the K-means clustering algorithm and applies it to the iris dataset.

It first defines a custom KMeans class that implements the K-means algorithm. The class has two main methods: fit, which trains the model on the provided data, and evaluate, which applies the trained model to new data and returns the assigned centroids and cluster assignments. The calculatewcss method is also defined to calculate the within-cluster sum of squared errors (WCSS) for the resulting clusters.

The KMeans class is instantiated with the desired number of clusters (centers=3) and trained on the iris dataset using the fit method. The evaluate method is then used to assign each sample to a cluster based on the learned centroids, and the resulting cluster assignments are compared to the true labels using the accuracyscore function from the sklearn.metrics module.

The `calculatewcss` method is also used to calculate the within-cluster sum of squared errors (WCSS) for the resulting clusters. The WCSS is a measure of the compactness of the clusters, with lower values indicating more tightly clustered data. The WCSS is printed to the console.



```
Accuracy is: 89.33 %
WCSS is: 78.85144142614601
```

4 Exercise 2.9 & 2.10

```
1 # Add a column of ones to the feature matrix for the bias term
2 X_train = np.hstack((np.ones((feature_train.shape[0], 1)), ←
    feature_train))
3
4 # Calculate the manual inverse matrix
5 XtX_inv = np.linalg.inv(np.dot(X_train.T, X_train))
6 Xty = np.dot(X_train.T, target_train)
7
8 # Calculate the weights using the manual inverse matrix
9 weights = np.dot(XtX_inv, Xty)
10
11 # Print the weights
12 print("Weights:")
13 print(weights)
14
15 # Predict on the training set
16 y_pred_train = np.dot(X_train, weights)
17
18 # Predict on the testing set
19 X_test = np.hstack((np.ones((feature_test.shape[0], 1)), feature_test)←
    )
20 y_pred_test = np.dot(X_test, weights)
21
22 # calculate mean squared error
23 print("MSE: " ,mean_squared_error(target_test, y_pred_test))
```

Explanation:

It performs linear regression on a given dataset using the normal equation method and calculates the mean squared error (MSE) for the predictions. The goal of it is to perform linear regression on a given dataset using the normal equation method and evaluate the performance of the resulting model by calculating the mean squared error (MSE) for the predictions.

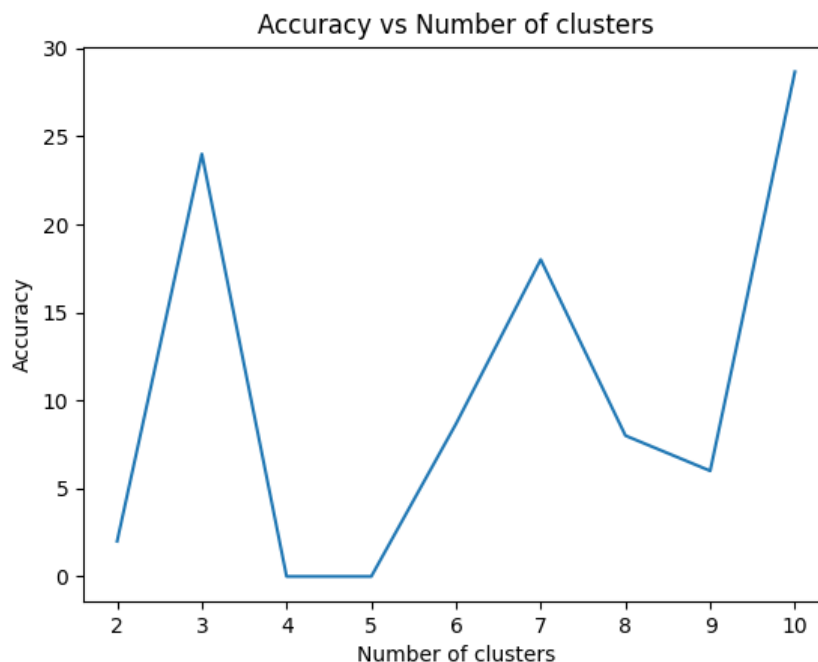
It adds a column of ones to the feature matrix for the bias term using the `hstack` function from

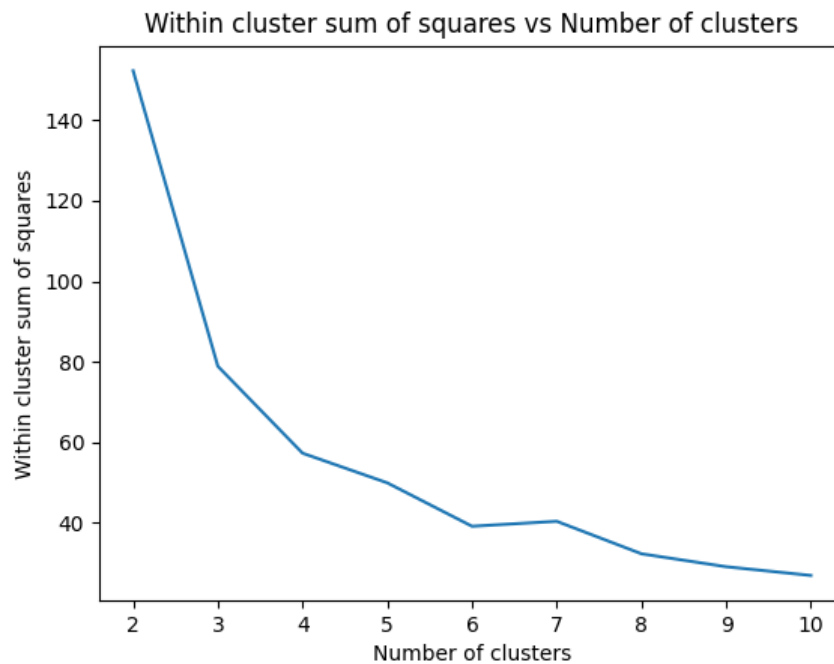
the numpy module. This is because the normal equation method requires the feature matrix to include a column of ones for the bias term.

It then calculates the manual inverse matrix using the inv function from the numpy.linalg module. This inverse matrix is then used to calculate the weights for the linear regression model using the dot product of the inverse matrix and the target values. The resulting weights are printed to the console using the print function from Python's standard library. These weights represent the coefficients for the linear regression model.

The learned weights are then used to make predictions on the training set and testing set. The testing set is transformed to include a column of ones for the bias term using the same technique as the training set. It calculates the MSE for the predictions using the meansquarederror function from the sklearn.metrics module and prints the result to the console.

```
Accuracy with clusters = 2 is: 2.0 %  
Accuracy with clusters = 3 is: 24.0 %  
Accuracy with clusters = 4 is: 0.0 %  
Accuracy with clusters = 5 is: 0.0 %  
Accuracy with clusters = 6 is: 8.67 %  
Accuracy with clusters = 7 is: 18.0 %  
Accuracy with clusters = 8 is: 8.0 %  
Accuracy with clusters = 9 is: 6.0 %  
Accuracy with clusters = 10 is: 28.67 %
```





5 Exercise 2.11

- Domain Knowledge: If you have prior knowledge or understanding of the dataset and the underlying problem, it can provide insights into the expected number of clusters. For example, if you are clustering customer data and you know there are distinct customer segments, you might have an idea of the approximate number of clusters.
- Elbow Method: The Elbow Method is a popular technique to determine the optimal value of k . It involves plotting the within-cluster sum of squares (WCSS) against different values of k and looking for the "elbow" point in the graph. The elbow point represents the value of k where the decrease in WCSS begins to level off significantly. This point indicates a trade-off between the number of clusters and the compactness of the data. The idea is to choose the k value at the elbow point since further increasing k may not provide significant improvement in clustering quality.
- Visualization: Plotting the data points and their features in a low-dimensional space (e.g., using dimensionality reduction techniques like PCA or t-SNE) can provide visual cues about the natural grouping of the data. This visualization can help you make an educated guess about the appropriate value of k .

6 Exercise 2.12 & 2.13 & 2.14 & 2.15 & 2.16

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn import datasets
```

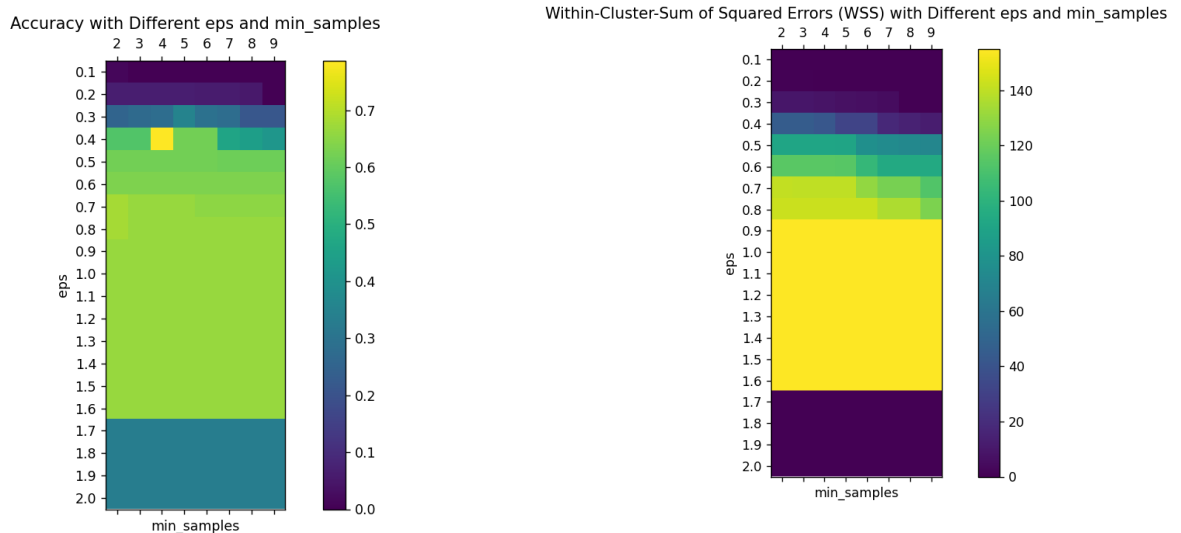
```
4 from sklearn.cluster import DBSCAN, KMeans
5 from sklearn.metrics import accuracy_score
6 from sklearn.model_selection import train_test_split
7
8 # Load the Iris dataset
9 iris = datasets.load_iris()
10 X = iris.data
11 y = iris.target
12
13 # Apply DBSCAN
14 dbscan = DBSCAN(eps=0.5, min_samples=5)
15 dbscan.fit(X)
16
17 # Evaluate accuracy
18 labels = dbscan.labels_
19 n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
20 accuracy = accuracy_score(y, labels)
21
22 print("DBSCAN Accuracy:", accuracy)
23
24 # Define range of eps and min_samples values
25 eps_values = np.linspace(0.1, 2.0, num=20)
26 min_samples_values = range(2, 10)
27
28 accuracy_results = np.zeros((len(eps_values), len(min_samples_values)))↵
29
30 # Perform DBSCAN with different eps and min_samples values
31 for i, eps in enumerate(eps_values):
32     for j, min_samples in enumerate(min_samples_values):
33         dbscan = DBSCAN(eps=eps, min_samples=min_samples)
34         dbscan.fit(X)
35         labels = dbscan.labels_
36         n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
37         accuracy = accuracy_score(y, labels)
38         accuracy_results[i, j] = accuracy
39
40 # Visualize the accuracy results
41 fig, ax = plt.subplots(figsize=(10, 6))
42 cax = ax.matshow(accuracy_results, cmap='viridis')
43 fig.colorbar(cax)
44 ax.set_xticks(np.arange(len(min_samples_values)))
45 ax.set_yticks(np.arange(len(eps_values)))
46 ax.set_xticklabels(min_samples_values)
47 ax.set_yticklabels(["{:.1f}".format(eps) for eps in eps_values])
```

```
48 ax.set_xlabel('min_samples')
49 ax.set_ylabel('eps')
50 ax.set_title('Accuracy with Different eps and min_samples')
51 plt.show()
52
53 # Calculate WSS values
54 wss_results = np.zeros((len(eps_values), len(min_samples_values)))
55
56 for i, eps in enumerate(eps_values):
57     for j, min_samples in enumerate(min_samples_values):
58         dbscan = DBSCAN(eps=eps, min_samples=min_samples)
59         dbscan.fit(X)
60         labels = dbscan.labels_
61         n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
62         if n_clusters > 1:
63             wss = 0
64             for k in range(n_clusters):
65                 cluster_points = X[labels == k]
66                 cluster_center = np.mean(cluster_points, axis=0)
67                 wss += np.sum((cluster_points - cluster_center) ** 2)
68             wss_results[i, j] = wss
69
70 # Visualize the WSS results
71 fig, ax = plt.subplots(figsize=(10, 6))
72 cax = ax.matshow(wss_results, cmap='viridis')
73 fig.colorbar(cax)
74 ax.set_xticks(np.arange(len(min_samples_values)))
75 ax.set_yticks(np.arange(len(eps_values)))
76 ax.set_xticklabels(min_samples_values)
77 ax.set_yticklabels(["{:.1f}".format(eps) for eps in eps_values])
78 ax.set_xlabel('min_samples')
79 ax.set_ylabel('eps')
80 ax.set_title('Within-Cluster-Sum of Squared Errors (WSS) with ↔
    Different eps and min_samples')
81 plt.show()
```

Explanation: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) finds core samples in regions of high density and expands clusters from them. This algorithm is good for data which contains clusters of similar density. From line 13-15, we use DBSCAN command with the following attributes $\text{eps} = 0.5$ and we take $\text{min} = 5$. To evaluate accuracy, we must take n_clusters and y value. To visualize the accuracy, we have to set x, y label (named min_samples and eps values). Line 54, we initialize 2D Numpy array called wss_results with zeros, the array has 2 attributes called eps values and min_samples values. Finally, we iterate through the eps values until wss for clusters is calculated with more than one point.

DBSCAN Accuracy: 0.62

Figure 2: Accuracy of DBSCAN



7 Exercise 2.17

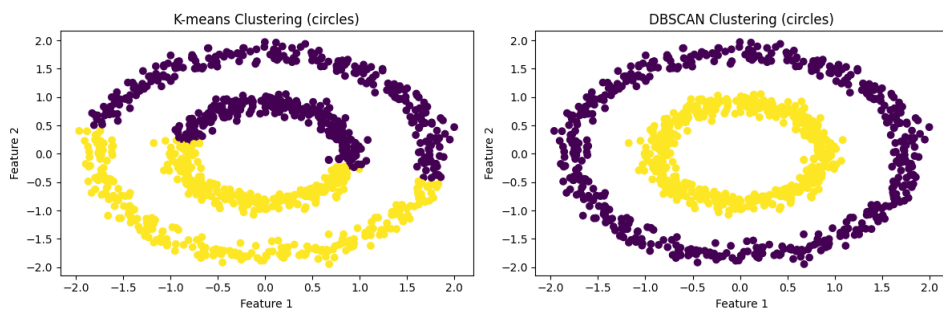
```

1 from sklearn.datasets import make_circles
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.cluster import DBSCAN, KMeans
4 import matplotlib.pyplot as plt
5 from sklearn.metrics import accuracy_score
6
7 # Generate circles dataset
8 X_circles, y_circles = make_circles(n_samples=1000, factor=0.5, noise←
    =0.05)
9
10 # Scale the features
11 scaler = StandardScaler()
12 X_circles = scaler.fit_transform(X_circles)
13
14 # Apply k-means
15 kmeans_circles = KMeans(n_clusters=2, random_state=42)
16 kmeans_circles.fit(X_circles)
17
18 # Apply DBSCAN
19 dbscan_circles = DBSCAN(eps=0.3, min_samples=5)
20 dbscan_circles.fit(X_circles)
21
22 # Plot k-means results

```

```
23 plt.figure(figsize=(12, 4))
24 plt.subplot(121)
25 plt.scatter(X_circles[:, 0], X_circles[:, 1], c=kmeans_circles.labels_↵
    , cmap='viridis')
26 plt.title('K-means Clustering (circles)')
27 plt.xlabel('Feature 1')
28 plt.ylabel('Feature 2')
29
30 # Plot DBSCAN results
31 plt.subplot(122)
32 plt.scatter(X_circles[:, 0], X_circles[:, 1], c=dbscan_circles.labels_↵
    , cmap='viridis')
33 plt.title('DBSCAN Clustering (circles)')
34 plt.xlabel('Feature 1')
35 plt.ylabel('Feature 2')
36
37 plt.tight_layout()
38 plt.show()
```

ExExplanation: Overall, using DBSCAN is much more efficient than K-means clustering because DBSCAN Clustering is better at dividing circles



8 Exercise 2.18

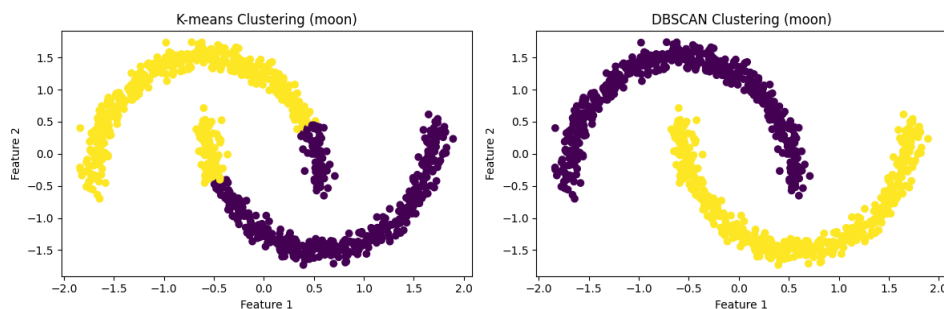
```
1 from sklearn.datasets import make_moons
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.cluster import DBSCAN, KMeans
4 import matplotlib.pyplot as plt
5 from sklearn.metrics import accuracy_score
6
7 # Generate the "moons" dataset
8 X_moons, y_moons = make_moons(n_samples=1000, noise=0.05)
9
10 # Scale the features
```

```

11 scaler = StandardScaler()
12 X_moons = scaler.fit_transform(X_moons)
13
14 # Apply k-means
15 kmeans_moon = KMeans(n_clusters=2, random_state=42)
16 kmeans_moon.fit(X_moons)
17
18 # Apply DBSCAN
19 dbscan_moon = DBSCAN(eps=0.3, min_samples=5)
20 dbscan_moon.fit(X_moons)
21
22 # Plot k-means results
23 plt.figure(figsize=(12, 4))
24 plt.subplot(121)
25 plt.scatter(X_moons[:, 0], X_moons[:, 1], c=kmeans_moon.labels_, cmap=↵
    'viridis')
26 plt.title('K-means Clustering (moon)')
27 plt.xlabel('Feature 1')
28 plt.ylabel('Feature 2')
29
30 # Plot DBSCAN results
31 plt.subplot(122)
32 plt.scatter(X_moons[:, 0], X_moons[:, 1], c=dbscan_moon.labels_, cmap=↵
    'viridis')
33 plt.title('DBSCAN Clustering (moon)')
34 plt.xlabel('Feature 1')
35 plt.ylabel('Feature 2')
36
37 plt.tight_layout()
38 plt.show()

```

Explanation: Overall, for moon datasets , DBSCAN is better than k-means in terms of clustering



9 Exercise 2.19

```
1 from sklearn.datasets import make_blobs
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.cluster import DBSCAN, KMeans
4 import matplotlib.pyplot as plt
5 from sklearn.metrics import accuracy_score
6
7 # Generate blobs dataset
8 X_blobs, y_blobs = make_blobs(n_samples=1000, centers=3, cluster_std=1.5)
9
10 # Scale the features
11 scaler = StandardScaler()
12 X_blobs = scaler.fit_transform(X_blobs)
13
14 # Apply k-means
15 kmeans_blobs = KMeans(n_clusters=3, random_state=42)
16 kmeans_blobs.fit(X_blobs)
17
18 # Apply DBSCAN
19 dbscan_blobs = DBSCAN(eps=0.3, min_samples=5)
20 dbscan_blobs.fit(X_blobs)
21
22 # Plot k-means results
23 plt.figure(figsize=(12, 4))
24 plt.subplot(121)
25 plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=kmeans_blobs.labels_, cmap=
    = 'viridis')
26 plt.title('K-means Clustering (blobs)')
27 plt.xlabel('Feature 1')
28 plt.ylabel('Feature 2')
29
30 # Plot DBSCAN results
31 plt.subplot(122)
32 plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=dbscan_blobs.labels_, cmap=
    = 'viridis')
33 plt.title('DBSCAN Clustering (blobs)')
34 plt.xlabel('Feature 1')
35 plt.ylabel('Feature 2')
36
37 plt.tight_layout()
38 plt.show()
```

Explanation : K-means clustering is better than DBSCAN in terms of dividing blobs into 3 small sections.

