1 Lab Notebook 15 - Clustering methods

In this notebook, we will learn how to write a **k-means++ clustering** code from scratch. We will test this code on two different data sets: one with spherically shaped clusters, and another with irregularly shaped clusters. Next, we will use **Density-Based Spatial Clustering of Applications with Noise (DBSCAN)** on the same data sets, and determine which ML method is better.

1.1 Step 1

Import the appropriate packages and, optionally, set your matplotlib.rc parameters.

In [1]:
```
import numpy as np
import pandas as pd
from scipy import stats
import matplotlib
import matplotlib.pyplot as plt
from sklearn import metrics

font = {'size'   : 16}
matplotlib.rc('font', **font)
matplotlib.rc('xtick', labelsize=14)
matplotlib.rc('ytick', labelsize=14)
matplotlib.rcParams['figure.dpi'] = 100
```

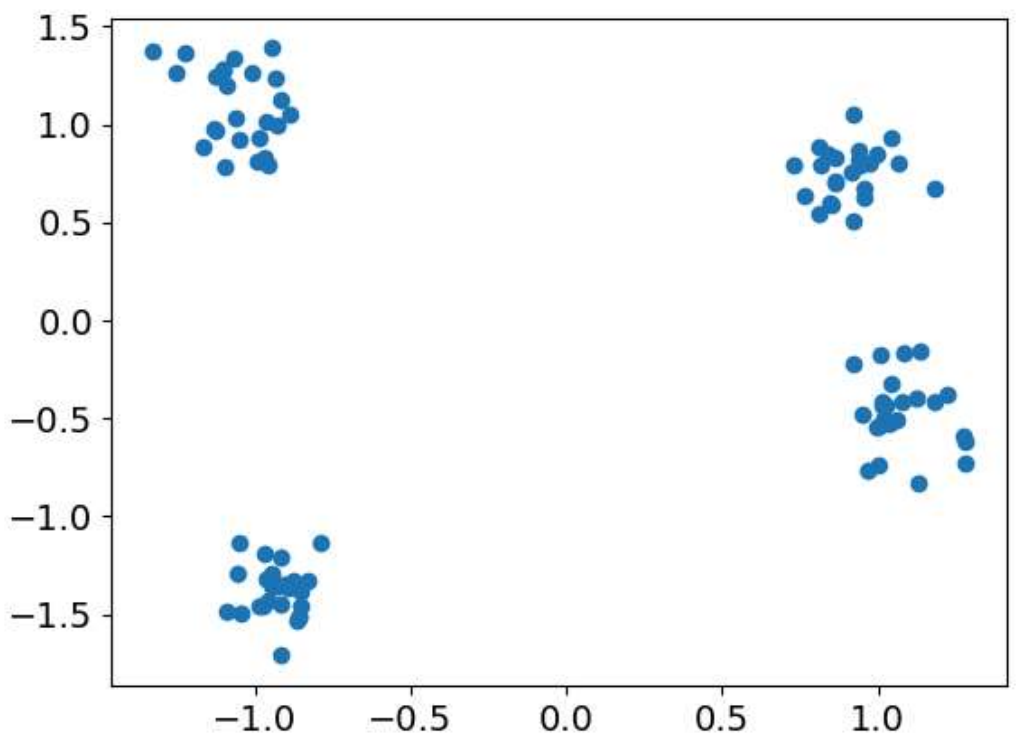
1.2 Step 2

Here is our first dataset. Make a scatter plot of the data points.

In [2]:
```
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Generate (approximately circular) data:
centers1 = 4
xvals1, true_labels1 = make_blobs(n_samples=100, centers=centers1, random_state=30,cluster_std=0.3)
xvals1 = StandardScaler().fit_transform(xvals1)
```

In [6]:
```
plt.scatter(xvals1[:,0],xvals1[:,1])
```

Out[6]: <matplotlib.collections.PathCollection at 0x22c0d37e740>

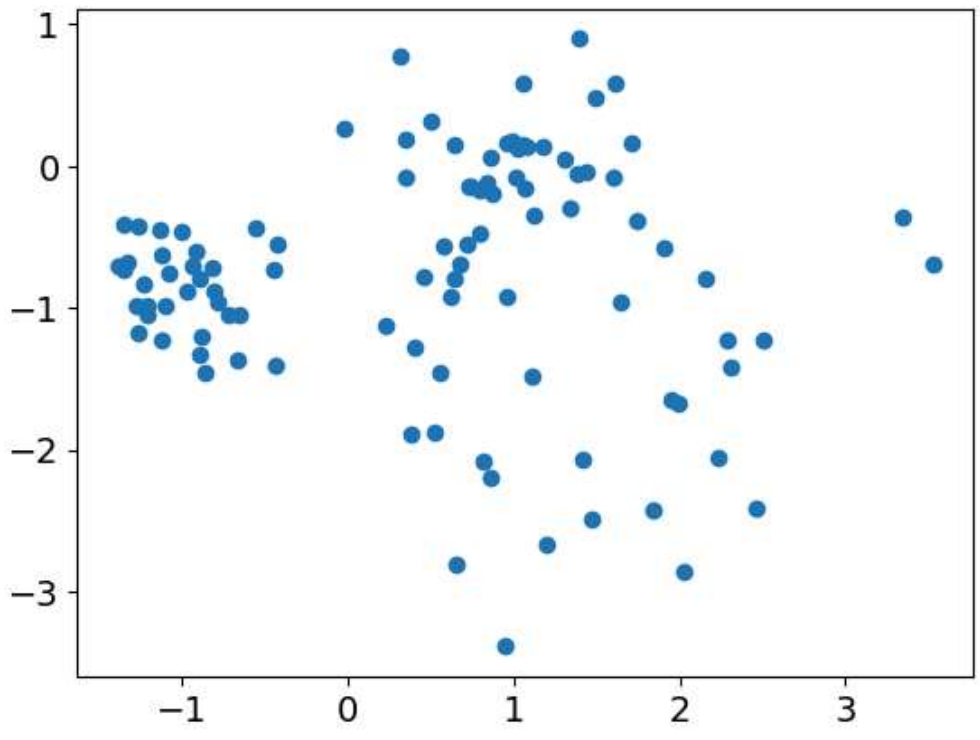


Here is our second dataset. Again make a scatter plot of the data.

In [9]:
```
# Generate (irregular) data:
centers2 = [[1, 0], [-1, -1], [1.5, -1.5]]
ncenters2 = 3
xvals2, true_labels2 = make_blobs(n_samples=100, centers=centers2, cluster_std=[0.4, 0.3, 0.85], random_state=0)
```

In [10]:
```
plt.scatter(xvals2[:,0],xvals2[:,1])
```

Out[10]: <matplotlib.collections.PathCollection at 0x22c0d448220>



1.3 Step 3

Now we will write our k-means++ clustering algorithm.

First, define a function called "euclidean" that intakes a point and a dataset of points, and returns the euclidean distance. The point has dimensions (m,), the data has dimensions (n,m), and output will be of size (n,). You already wrote such a function for the knn algorithm in lab 3, feel free to import your code.

In [11]:
```
def euclidean(point, data):
    diff = data - point
    sq_dist = np.sum(diff**2, axis=1)
    dist = np.sqrt(sq_dist)
    return dist
```

Next, define the class "KMeans". Within this class, we have the following functions:

1. **__init__**, which has inputs self, n_clusters, and max_iter. You can set default values for n_clusters and max_iter if you wish. Let self.n_clusters = n_clusters and self.max_iter = max_iter.
2. **fit**, which has inputs self and X_train. This is the core of the kmeans algorithm.
3. **evaluate**, which intakes self and X. This function will evaluate the distance between a set of points and the centroids for which we've optimized our training set. This method returns classification labels, i.e. the index of the centroid to which each data point belongs.

init and fit are already provided. You need to complete a few lines in **evaluate**

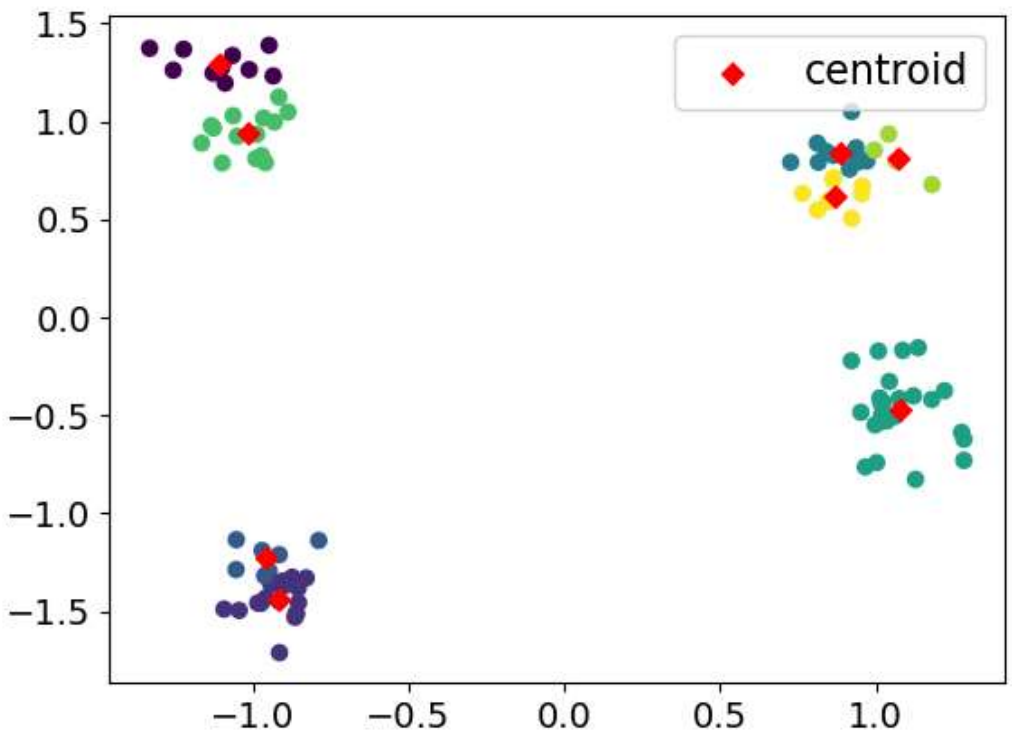
In [15]:
```
from sklearn.preprocessing import StandardScaler
from numpy.random import uniform
import random

class KMeans:
    def __init__(self, n_clusters=8, max_iter=300):
        self.n_clusters = n_clusters
        self.max_iter = max_iter

    def fit(self, X_train):
        # We provide the following code, also called the "k-means++" method, to initialize the centroids.
        # A random datapoint is selected as the first,
        # then the rest are initialized w/ probabilities proportional to their distances to the first point

        # Pick a random point from train data for first centroid
        self.centroids = [random.choice(X_train)]
        # Set the remaining initial guesses
        for _ in range(self.n_clusters-1):
            # Calculate distances from points to the centroids
            dists = np.sum([euclidean(centroid, X_train) for centroid in self.centroids], axis=0)
            # Normalize the distances
            dists /= np.sum(dists)
            # Choose remaining points based on their distances
            new_centroid_idx, = np.random.choice(range(len(X_train)), size=1, p=dists)
            self.centroids += [X_train[new_centroid_idx]]

        # end of initialization

        # This initial method of randomly selecting centroid starts is less effective
        # min_, max_ = np.min(X_train, axis=0), np.max(X_train, axis=0)
        # self.centroids = [uniform(min_, max_) for _ in range(self.n_clusters)]

        # Iterate, adjusting centroids until converged or until passed max_iter
        iteration = 0
        # initialize a counter for the number of iterations
        prev_centroids = None
        #intialize an array prev_centroids = None
        # while centroids are changing and max_iter is not exceeded:

        while np.not_equal(self.centroids, prev_centroids).any() and iteration < self.max_iter:
            # initialize and empty array sorted_points
            sorted_points = [[] for _ in range(self.n_clusters)]

            # loop over training set, compute distances data-centroids, find closest centroid for each data point,
            # append data point to list belonging to that centroid
            for x in X_train:
                dists = euclidean(x, self.centroids)
                centroid_idx = np.argmin(dists)
                sorted_points[centroid_idx].append(x)

            # Store current centroids in previous, reassign centroids as mean of the points belonging to them
            prev_centroids = self.centroids
            self.centroids = [np.mean(cluster, axis=0) for cluster in sorted_points]

            # Restore previous centroid if new one has no points
            for i, centroid in enumerate(self.centroids):
                if np.isnan(centroid).any():  # Catch any np.nans, resulting from a centroid having no points
                    self.centroids[i] = prev_centroids[i]
            # Increment counter
            iteration += 1

    def evaluate(self, X):
        #initialize an empty list for the indices of the centroids
        centroid_idxs = []
        #loop over all examples
        for x in X:
            # compute distances to all centroids
            distances = euclidean(x,self.centroids)
            # find the clostest centroid for each data point
            closest_centroid = min(distances)
            # append index to list of indices
            min_index = np.where(distances == closest_centroid)[0][0]
            centroid_idxs.append(min_index)
        # return centroid indices
        return centroid_idxs
```

1.4 Step 4

Test the code on dataset 1. Make a scatter plot of the data, color the data by classification label, and draw the positions of the centroids in a different symbol and color.

In [33]:
```
kmeans= KMeans()
kmeans.fit(xvals1)

y_pred = kmeans.evaluate(xvals1)

# Define a list of colors. We will use a matplotlib colormap to generate 8 distinct colors.
colormap = plt.cm.get_cmap('viridis', 8)  # 'tab10' colormap has 10 distinct colors

# Create a mapping from labels to colors
label_to_color = {label: colormap(label) for label in np.unique(y_pred)}

# Map the labels to the colors
mapped_colors = np.array([label_to_color[label] for label in y_pred])

centroids = np.array(kmeans.centroids)
# Now you can use 'mapped_colors' for the color argument in plt.scatter
plt.scatter(xvals1[:, 0], xvals1[:, 1], c=mapped_colors)
plt.scatter(centroids[:,0],centroids[:,1],marker = "D" , c = "Red", label = "centroid")
plt.legend()
plt.show()
```

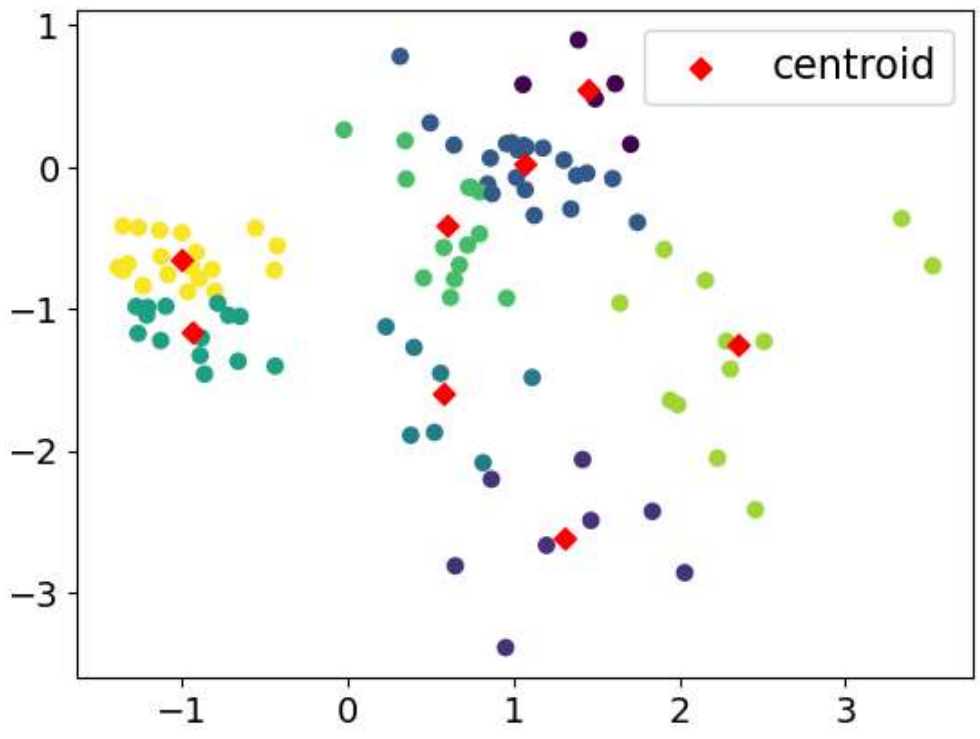
```
C:\Users\kesha\AppData\Local\Temp\ipykernel_30500\2798698945.py:7: MatplotlibDeprecationWarning: The get_cmap function was
deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplot
lib.colormaps.get_cmap(obj)`` instead.
  colormap = plt.cm.get_cmap('viridis', 8)  # 'tab10' colormap has 10 distinct colors
```



Repeat the above on dataset 2.

In [36]:
```
kmeans.fit(xvals2)
y_pred = kmeans.evaluate(xvals2)

# Define a list of colors. We will use a matplotlib colormap to generate 8 distinct colors.
colormap = plt.cm.get_cmap('viridis', 8)  # 'tab10' colormap has 10 distinct colors

# Create a mapping from labels to colors
label_to_color = {label: colormap(label) for label in np.unique(y_pred)}

# Map the labels to the colors
mapped_colors = np.array([label_to_color[label] for label in y_pred])

centroids = np.array(kmeans.centroids)
# Now you can use 'mapped_colors' for the color argument in plt.scatter
plt.scatter(xvals2[:, 0], xvals2[:, 1], c=mapped_colors)
plt.scatter(centroids[:,0],centroids[:,1],marker = "D" , c = "Red", label = "centroid")
plt.legend()
plt.show()
```

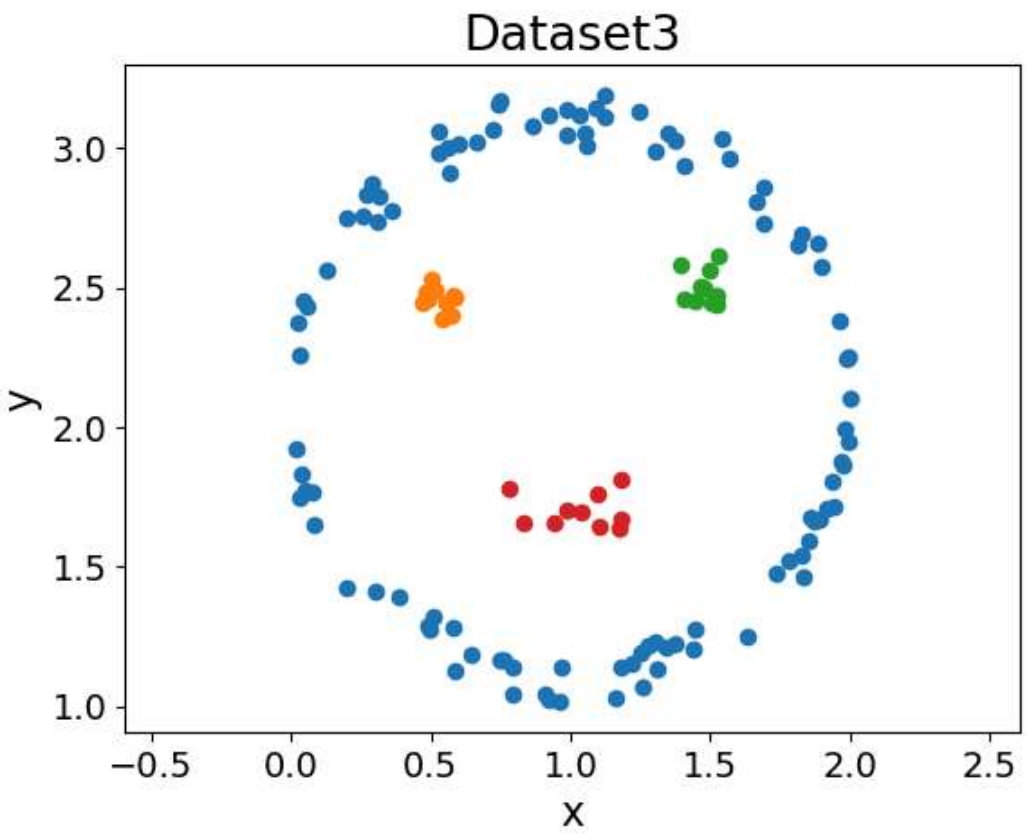
```
C:\Users\kesha\AppData\Local\Temp\ipykernel_30500\3473810041.py:5: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  colormap = plt.cm.get_cmap('viridis', 8)  # 'tab10' colormap has 10 distinct colors
```



1.5 Step 5

Hopefully, your k-means+ code worked to find the centroids of Datasets 1 and 2. However, in real life your datasets will likely be more complicated. For instance, see the following "smiley face" data:

In [23]:
```
from math import pi, cos, sin
import random
np.random.seed(10)
def point(h, k, r):
    theta = random.random() * 2 * pi
    return h + cos(theta) * r, k + sin(theta) * r + 0.2*random.random()

# Generate points:
xy = [point(1,2,1) for _ in range(100)]

X1, y1 = make_blobs(n_samples=10, centers=[(0.5,2.5)],
                        cluster_std=0.05, random_state=1)

X2, y2 = make_blobs(n_samples=10, centers=[(1.5,2.5)],
                        cluster_std=0.05, random_state=2)

X3, y3 = make_blobs(n_samples=10, centers=[(1,1.7)],
                        cluster_std=0.05, random_state=2)

X3_stretch = np.array([X3[:,0]*3, X3[:,1]]) #make the mouth :)


# Plot:
plt.axes().set_aspect('equal', 'datalim')
plt.scatter(*zip(*xy))
plt.scatter(X1[:,0],X1[:,1])
plt.scatter(X2[:,0],X2[:,1])
plt.scatter(X3_stretch.T[:,0]-1.9,X3_stretch.T[:,1])

plt.title("Dataset3")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

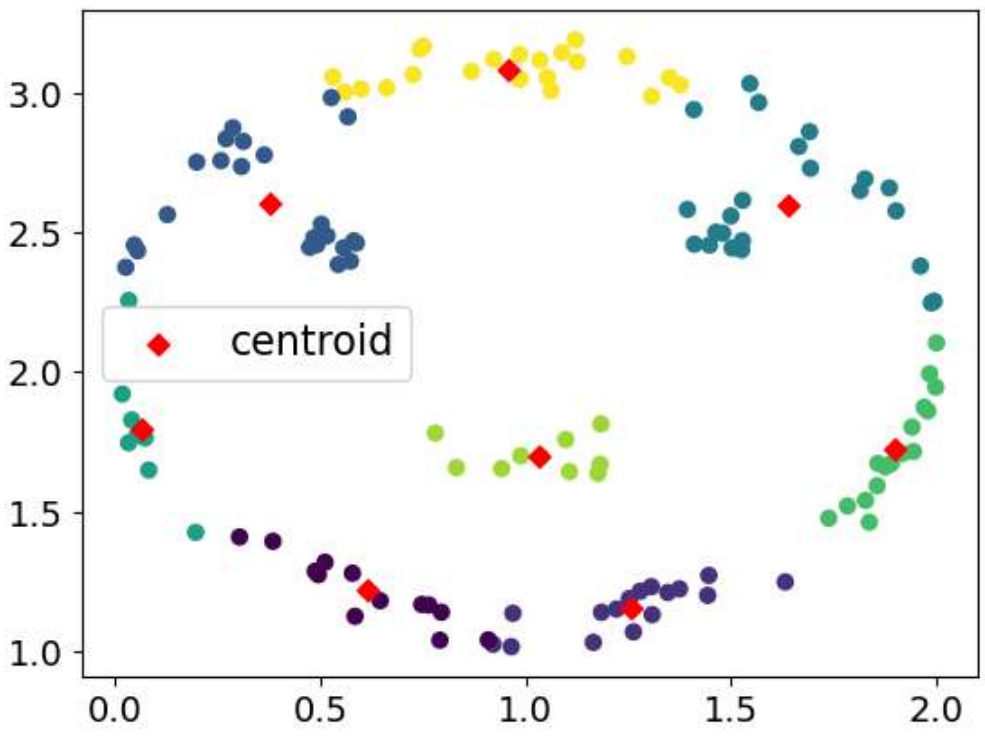


Try running your code on the "smiley face" dataset. Does your code accurately predict the centroids? You will need to use the following line of code (or something similar) to turn the smiley face data arrays into one array, "xvals3":

```
xvals3 = np.vstack([xy,X1,X2,np.array([X3_stretch.T[:,0]-1.9,X3_stretch.T[:,1]]).T])
```

In [38]:
```
xvals3 = np.vstack([xy,X1,X2,np.array([X3_stretch.T[:,0]-1.9,X3_stretch.T[:,1]]).T])

kmeans.fit(xvals3)
y_pred = kmeans.evaluate(xvals3)

# Define a list of colors. We will use a matplotlib colormap to generate 8 distinct colors.
colormap = plt.cm.get_cmap('viridis', 8)  # 'tab10' colormap has 10 distinct colors

# Create a mapping from labels to colors
label_to_color = {label: colormap(label) for label in np.unique(y_pred)}

# Map the labels to the colors
mapped_colors = np.array([label_to_color[label] for label in y_pred])

centroids = np.array(kmeans.centroids)
# Now you can use 'mapped_colors' for the color argument in plt.scatter
plt.scatter(xvals3[:, 0], xvals3[:, 1], c=mapped_colors)
plt.scatter(centroids[:,0],centroids[:,1],marker = "D" , c = "Red", label = "centroid")
plt.legend()
plt.show()
```

```
C:\Users\kesha\AppData\Local\Temp\ipykernel_30500\2916221260.py:7: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  colormap = plt.cm.get_cmap('viridis', 8)  # 'tab10' colormap has 10 distinct colors
```



Did the k-means++ code accurately predict the centroids? Why or why not?

1.6 Answer:

Qualitatively the kmeans code doesn't accurately identify the clusters. This is because we initialized the algorithm to look for 8 clusters but the dataset itself has 5 clusters, so there are 3 other clusters forced in.

1.7 Step 6

Another method we can use is called **Density-Based Spatial Clustering of Applications with Noise (DBSCAN)**. For a a set of points in space, this algorithm groups the points that are closely packed together, and identifies points that lie in low-density regions as outliers.

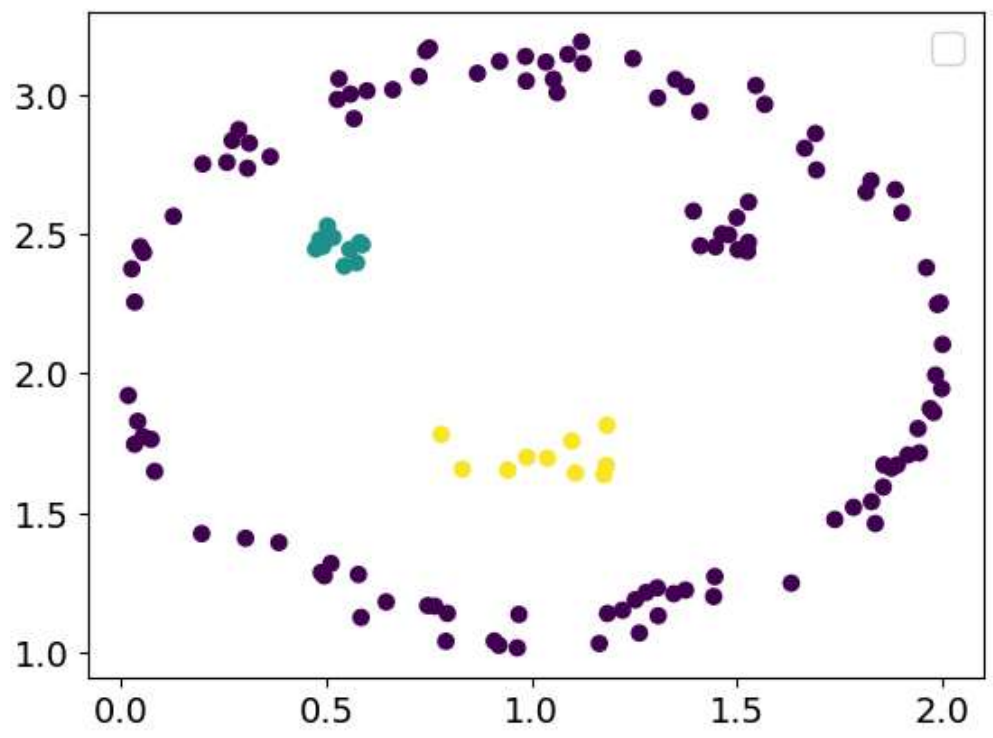
From sklearn.cluster, import "DBSCAN". We will be using the "smiley face" data to test this method. Plot the data colored by cluster label. How many clusters does DBSCAN find? How does the answer depend on the parameter **eps**?

In [79]:
```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# Create DBSCAN instance
dbscan = DBSCAN(eps=0.252, min_samples=2)

# Fit the model
dbscan.fit(xvals3)

# Get the cluster labels (Note: '-1' means an outlier)
labels = dbscan.labels_

# The core samples are the points that form the core of the clusters
core_samples_mask = np.zeros_like(dbscan.labels_, dtype=bool)
core_samples_mask[dbscan.core_sample_indices_] = True

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

labels, n_clusters_
```

Out[79]:
```
(array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       dtype=int64),
 3)
```

In [80]:
```
# Define a list of colors. We will use a matplotlib colormap to generate 8 distinct colors.
colormap = plt.cm.get_cmap('viridis', n_clusters_)  # 'tab10' colormap has 10 distinct colors

# Create a mapping from labels to colors
label_to_color = {label: colormap(label) for label in np.unique(labels)}

# Map the labels to the colors
mapped_colors = np.array([label_to_color[label] for label in labels])

centroids = np.array(kmeans.centroids)
# Now you can use 'mapped_colors' for the color argument in plt.scatter
plt.scatter(xvals3[:, 0], xvals3[:, 1], c=mapped_colors)
plt.legend()
plt.show()
```

```
C:\Users\kesha\AppData\Local\Temp\ipykernel_30500\360252063.py:2: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  colormap = plt.cm.get_cmap('viridis', n_clusters_)  # 'tab10' colormap has 10 distinct colors
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



1.8 Answer:

We notice that when reducing the value of eps, we get many clusters. The optimal value for this particular data distribution is a value for eps that is 0.252