Finding insights and value in data is the ambitious promise that has been seen in the rise of machine learning.

Unsupervised learning is the field of practice that helps find patterns in cluttered data and is one of the most exciting areas of development in machine learning today.

Clustering is the overarching process that involves finding groups of similar data that exist in your dataset, which can be extremely valuable if you are trying to find its underlying meaning.

One of the most basic yet popular approaches is to use a cluster analysis technique called **k-means clustering**. The k-means clustering works by searching for k clusters in your data and the workflow is actually quite intuitive.

Exercise 1: K-means from Scratch – Part 1: Data Generation

This exercise relies on scikit-learn, an open source Python package that enables the fast prototyping of popular machine learning models. Within scikit-learn, we will be using the `datasets` functionality to create a synthetic blob dataset. In addition to harnessing the power of scikit-learn, we will also

rely on Matplotlib, a popular plotting library for Python that makes it easy for us to visualize our data. To do this, perform the following steps:

1. Import the necessary libraries:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np
import math
np.random.seed(0)
%matplotlib inline
```

Note

You can find more details on the **`KMeans`** *library at https://scikit-learn.org/stable/modules/clustering.html#k-means.*

2. Generate a random cluster dataset to experiment on X = coordinate points, y = cluster labels, and define random centroids. We will achieve this with the **`make_blobs`** function that we imported from **`sklearn.datasets`**, which, as the name implies, generates blobs of data points.

```
X, y = make_blobs(n_samples=1500,
centers=3, \
                  n_features=2,
random_state=800)
centroids = [[-6,2],[3,-4],[-5,10]]
```

Here the **n_samples** parameter determines the total number of data points generated by the blobs. The **centers** parameter determines the number of centroids for the blob. The **n_feature** attribute defines the number of dimensions generated by the dataset. Here, the data will be two dimensional.

In order to generate the same data points in all the iterations (which in turn are generated randomly) for reproducibility of results, we set the **random_state** parameter to **800**. Different values of the **random_state** parameter would yield different results. If we do not set the **random_state** parameter, each time on execution we will obtain different results.

3. Print the data:

```
X
```

The output is as follows:

```
array([[-3.83458347, 6.09210705],
       [-4.62571831, 5.54296865],
       [-2.87807159, -7.48754592],
       ...,
       [-3.709726 , -7.77993633],
       [-8.44553266, -1.83519866],
       [-4.68308431, 6.91780744]])
```

4. Plot the coordinate points using the scatterplot functionality we imported from **matplotlib.pyplot**. This function

takes input lists of points and presents them graphically for ease of understanding. Please review the **`matplotlib`** documentation if you want to explore the parameters provided at a deeper level:

```
plt.scatter(X[:, 0], X[:, 1], s=50,
cmap='tab20b')
plt.show()
```

The plot appears as follows:

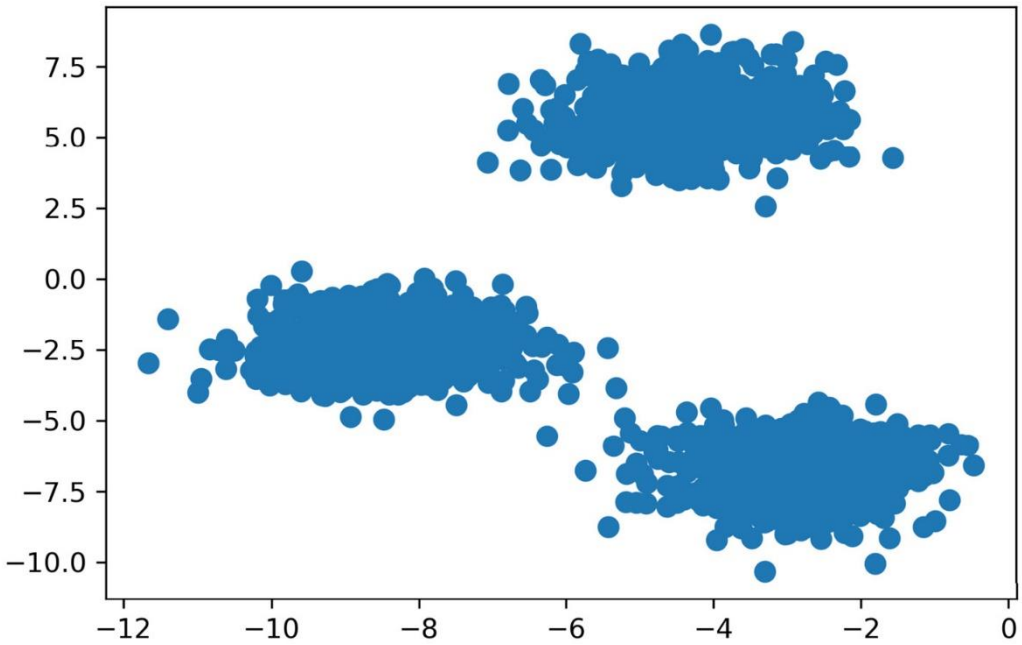

Figure 1: Plot of the coordinates

5. Print the array of **`y`**, which is the labels provided by scikit-learn and serves as the ground truth for comparison.

   Note

   These labels will not be known to us in practice. This is just for us to cross verify our clustering in later stages.

   Use the following code to print the array:

   ```
   y
   ```

The output is as follows:

```
array([2, 2, 1, ..., 1, 0, 2])
```

6. Plot the coordinate points with the correct cluster labels:

```
plt.scatter(X[:, 0], X[:, 1], c=y,s=50,
cmap='tab20b')
plt.show()
```

The plot appears as follows:

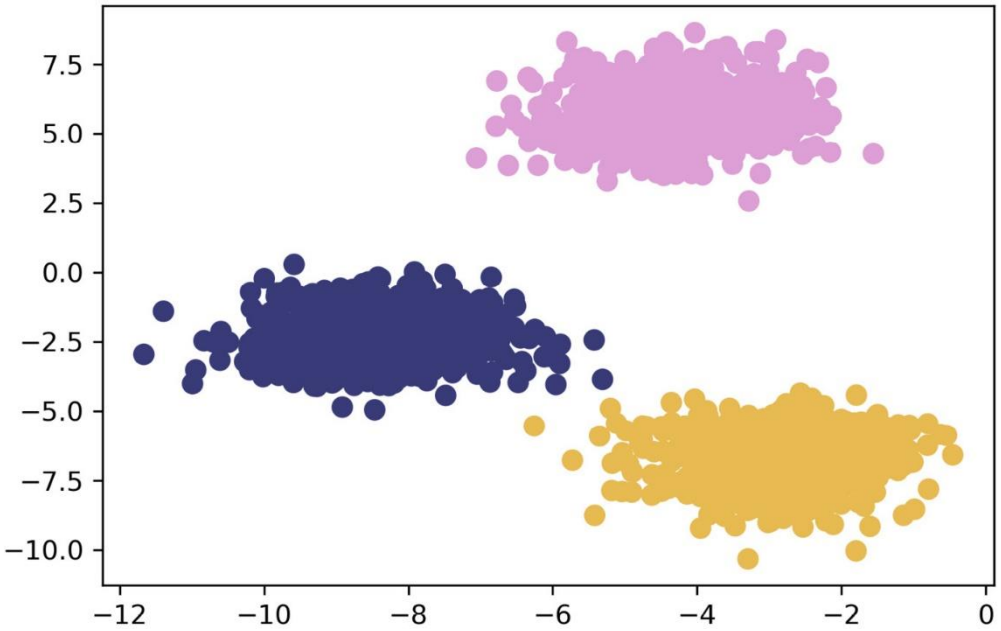

Figure 2: Plot of the coordinates with correct cluster labels

By completing the preceding steps, you have generated the data and visually explored how it is put together. By visualizing the ground truth, you have established a baseline that provides a relative metric for algorithm accuracy.

With data in hand, in the next exercise, we'll continue by building your unsupervised learning

toolset with an optimized version of the Euclidean distance function from the `SciPy` package, `cdist`. You will compare a non-vectorized, clearly understandable version of the approach with `cdist`, which has been specially tweaked for maximum performance.

Exercise 2 K-means from Scratch – Part 2: Implementing k-means

Let's recreate these results on our own. We will go over an example implementing this with some optimizations.

Note

This exercise is a continuation of the previous exercise and should be performed in the same Jupyter notebook.

For this exercise, we will rely on SciPy, a Python package that allows easy access to highly optimized versions of scientific calculations. In particular, we will be implementing Euclidean distance with `cdist`, the functionally of which replicates the barebones implementation of our distance metric in

a much more efficient manner. Follow these steps to complete this exercise:

1. The basis of this exercise will be comparing a basic implementation of Euclidean distance with an optimized version provided in SciPy. First, import the optimized Euclidean distance reference:
```
from scipy.spatial.distance import cdist
```
2. Identify a subset of **x** you want to explore. For this example, we are only selecting five points to make the lesson clearer; however, this approach scales to any number of points. We chose points 105-109, inclusive:
```
X[105:110]
```

The output is as follows:
```
array([[-3.09897933, 4.79407445],
       [-3.37295914, -7.36901393],
       [-3.372895 , 5.10433846],
       [-5.90267987, -3.28352194],
       [-3.52067739, 7.7841276 ]])
```

3. Calculate the distances and choose the index of the shortest distance as a cluster:
```
"""
Finds distances from each of 5 sampled
points to all of the centroids
"""
for x in X[105:110]:
    calcs = cdist(x.reshape([1,-
1]),centroids).squeeze()
```

```
    print(calcs, "Cluster Membership: ",
np.argmin(calcs))
```

Note

The triple-quotes (""") shown in the code snippet above are used to denote the start and end points of a multi-line code comment. Comments are added into code to help explain specific bits of logic.

The preceding code will result in the following output:

```
[4.027750355981394, 10.70202290628413,
5.542160268055164]
 Cluster Membership: 0
[9.73035280174993, 7.208665829113462,
17.44505393393603]
 Cluster Membership: 1
[4.066767506545852, 11.113179986633003,
5.158970112430151 5]
 Cluster Membership: 0
[5.284418164665783, 8.931464028407861,
13.314157359115697]
 Cluster Membership: 0
[6.293105164930943, 13.467921029846712,
2.664298385076878]
 Cluster Membership: 2
```

4. Define the **`k_means`** function as follows and initialize the k-centroids randomly. Repeat this process until the difference between the new/old **`centroids`** equals `0`, using the **`while`** loop:

```
def k_means(X, K):
    # Keep track of history so you can
see K-Means in action
    centroids_history = []
    labels_history = []
    rand_index =
np.random.choice(X.shape[0], K)
    centroids = X[rand_index]
    centroids_history.append(centroids)
```

5. Zip together the historical steps of centers and their labels:

```
history = zip(centers_hist, labels_hist)
for x, y in history:
    plt.figure(figsize=(4,3))
    plt.scatter(X[:, 0], X[:, 1], c=y,
s=50, cmap='tab20b');
    plt.scatter(x[:, 0], x[:, 1],
c='red')
    plt.show()
```

The following plots may differ from what you can see if we haven't set the random seed. The first plot looks as follows:

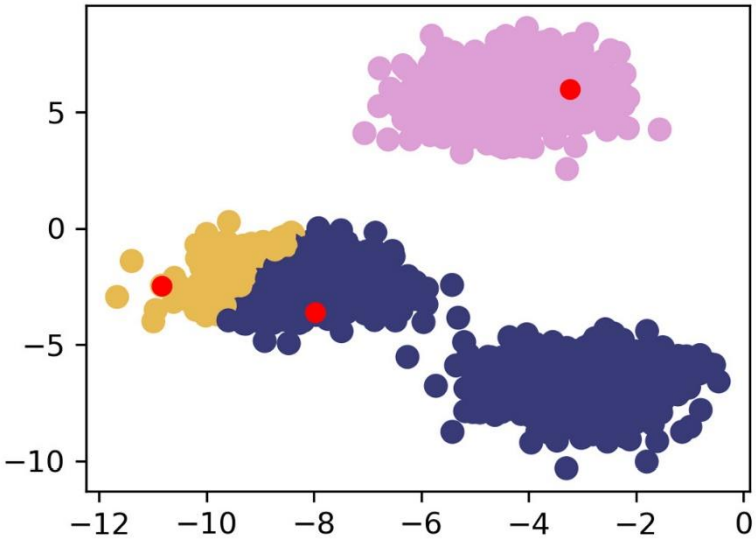

Figure 3: First scatterplot

The second plot appears as follows:

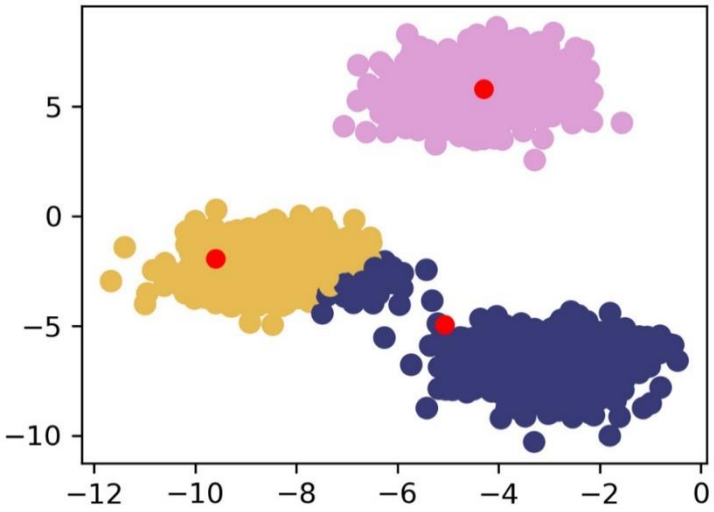

Figure 4: Second scatterplot

The third plot appears as follows:

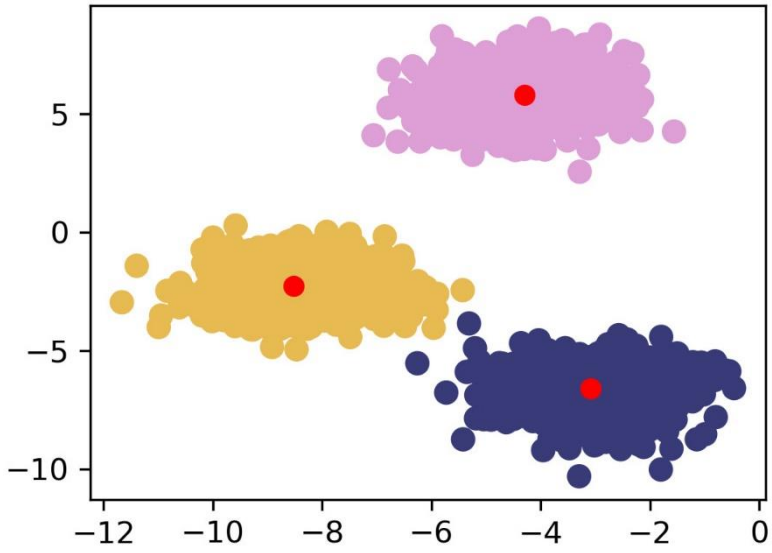

Figure 5: Third scatterplot

The fourth plot appears as follows:

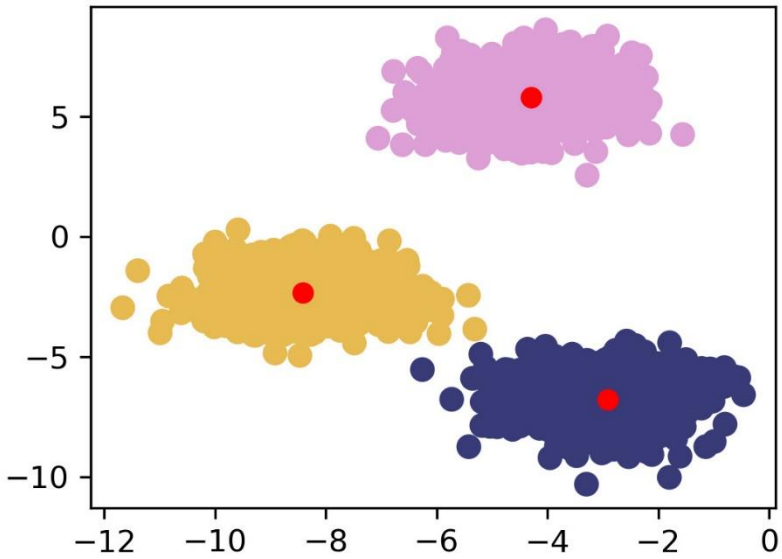

Figure 6: Fourth scatterplot

The fifth plot looks as follows:

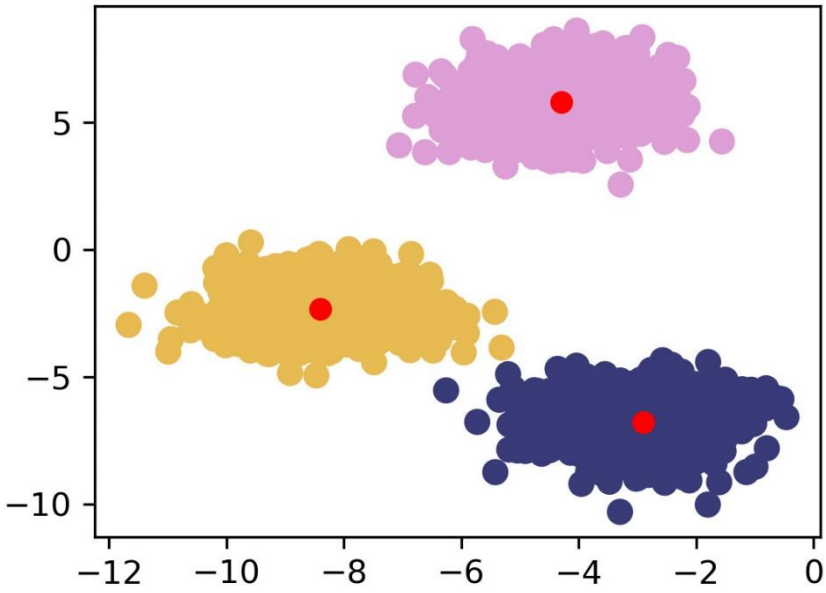

Figure 7: Fifth scatterplot

As shown by the preceding images, k-means takes an iterative approach to refine optimal clusters based on distance. The algorithm starts with random initialization of centroids and, depending on the complexity of the data, quickly finds the separations that make the most sense.