Cluster Analysis in Python

Grouping similar items together

Black Raven (James Ng)

14 Nov 2020 · 29 min read

This is a memo to share what I have learnt in Cluster Analysis (in Python), capturing the learning objectives as well as my personal notes. The course is taught by Shaumik Daityari from DataCamp, and it includes 4 chapters:

Chapter 1. Introduction to Clustering

Chapter 2. Hierarchical Clustering

Chapter 3. K-Means Clustering

Chapter 4. Clustering in Real World

You have probably come across Google News, which automatically groups similar news articles under a topic. Have you ever wondered what process runs in the background to arrive at these groups? In this course, you will be introduced to unsupervised learning through clustering using the SciPy library in Python. This course covers pre-processing of data and application of hierarchical and k-means clustering. Through the course, you will explore player statistics from a popular football video game, FIFA 18. After completing the course, you will be able to quickly apply various clustering algorithms on data, visualize the clusters formed and analyze results.

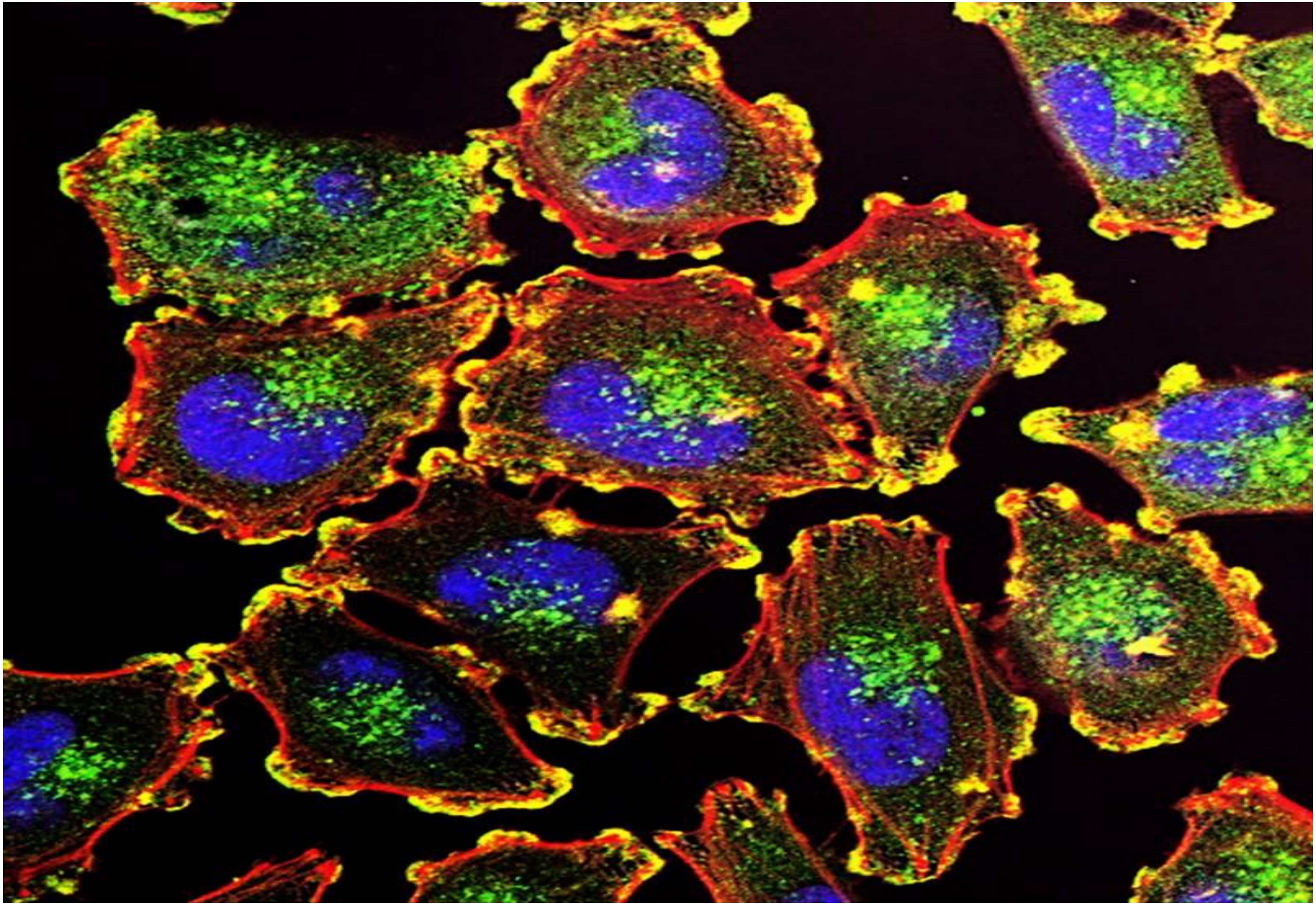


Photo by National Cancer Institute on Unsplash

Chapter 1. Clustering for dataset exploration

Before you are ready to classify news articles, you need to be introduced to the basics of clustering. This chapter familiarizes you with a class of machine learning algorithms called unsupervised learning and then introduces you to clustering, one of the popular unsupervised learning algorithms. You will know about two popular clustering techniques - hierarchical clustering and k-means clustering. The chapter concludes with basic pre-processing steps before you start clustering data.

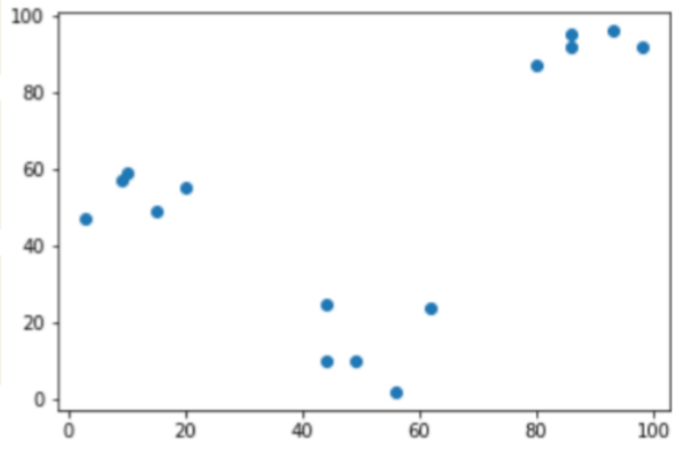
Unsupervised learning: basics

Google News classify similar articles together by unsupervised learning algorithm. It scans through the text of each article, and based on frequently occuring terms, groups articles together.

```python
from matplotlib import pyplot as plt
```

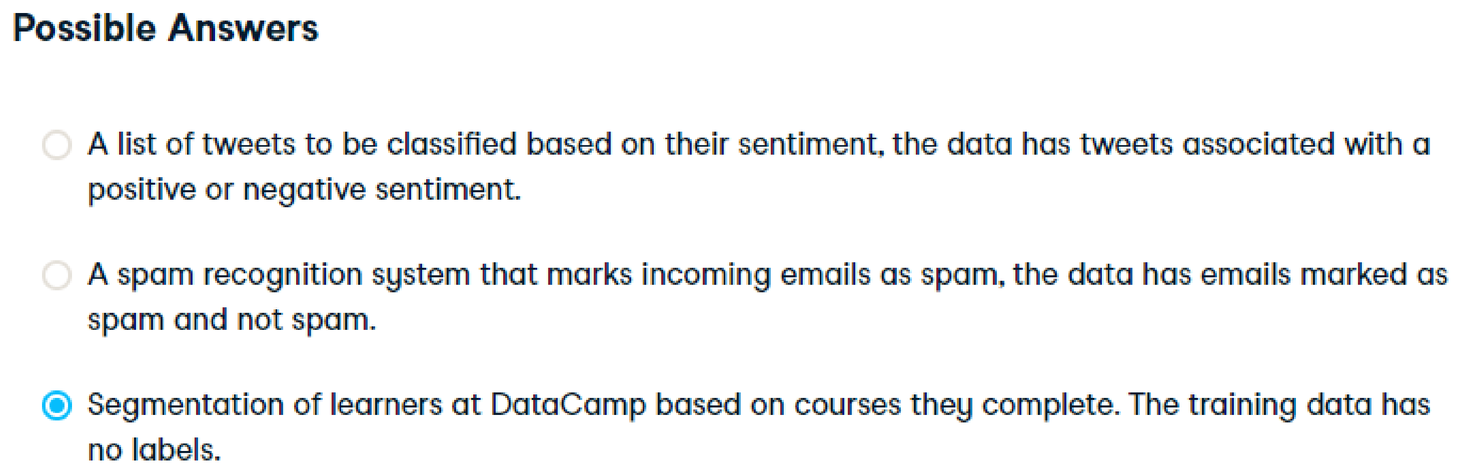
```python
x_coordinates = [80, 93, 86, 98, 86, 9, 15, 3, 10, 20, 44, 56, 49, 62, 44]
y_coordinates = [87, 96, 95, 92, 92, 57, 49, 47, 59, 55, 25, 2, 10, 24, 10]
```

```python
plt.scatter(x_coordinates, y_coordinates)
plt.show()
```



Unsupervised learning in real world

Which of the following examples can be solved with unsupervised learning?



Possible Answers

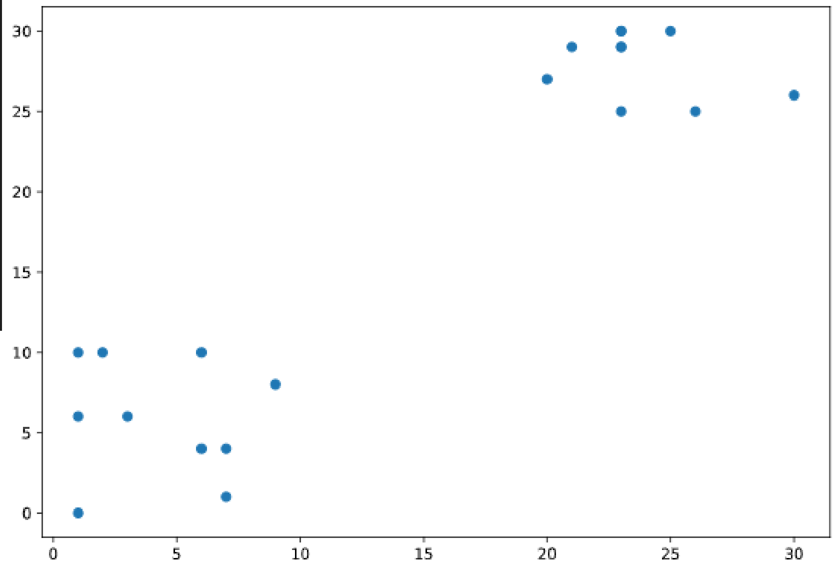
- A list of tweets to be classified based on their sentiment, the data has tweets associated with a positive or negative sentiment.
- A spam recognition system that marks incoming emails as spam, the data has emails marked as spam and not spam.
- [✓ Segmentation of learners at DataCamp based on courses they complete. The training data has no labels.]

As the training data has no labels, an unsupervised algorithm needs to be used to understand patterns in the data.

Pokémon sightings

There have been reports of sightings of rare, legendary Pokémon. You have been asked to investigate! Plot the coordinates of sightings to find out where the Pokémon might be. The X and Y coordinates of the points are stored in list `x` and `y`, respectively.

```python
# Import plotting class from matplotlib library
from matplotlib import pyplot as plt

# Create a scatter plot
plt.scatter(x, y)

# Display the scatter plot
plt.show()
```



Notice the areas where the sightings are dense. This indicates that there is not one, but two legendary Pokémon out there!

Basics of cluster analysis

Cluster = group of items with similar characteristics
Eg. news articles with similar words and word associations, customer segments with similar spending habits

Clustering algorithms: Hierarchical clustering, K-means clustering, DBSCAN, Gaussian Methods

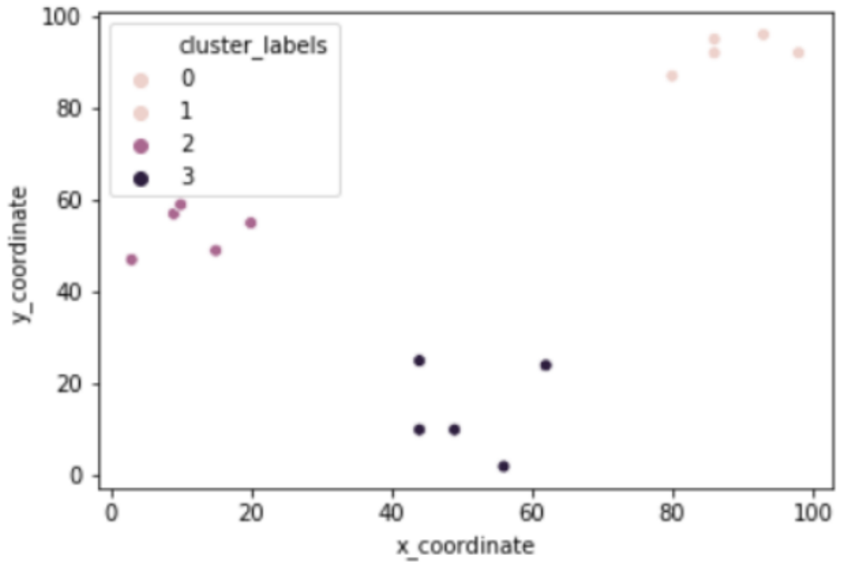
Hierarchical clustering

```python
from scipy.cluster.hierarchy import linkage, fcluster
from matplotlib import pyplot as plt
import seaborn as sns, pandas as pd
```

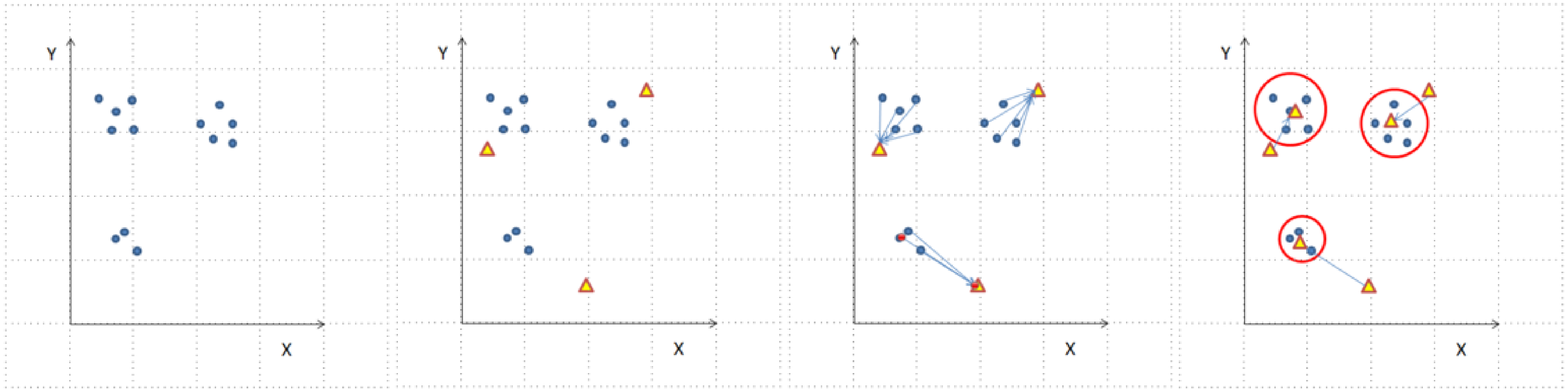
```python
x_coordinates = [80.1, 93.1, 86.6, 98.5, 86.4, 9.5, 15.2, 3.4,
                 10.4, 20.3, 44.2, 56.8, 49.2, 62.5, 44.0]
y_coordinates = [87.2, 96.1, 95.6, 92.4, 92.4, 57.7, 49.4,
                 47.3, 59.1, 55.5, 25.6, 2.1, 10.9, 24.1, 10.3]

df = pd.DataFrame({'x_coordinate': x_coordinates,
                   'y_coordinate': y_coordinates})
```

```python
Z = linkage(df, 'ward')
df['cluster_labels'] = fcluster(Z, 3, criterion='maxclust')
```

```python
sns.scatterplot(x='x_coordinate', y='y_coordinate',
                hue='cluster_labels', data = df)
plt.show()
```



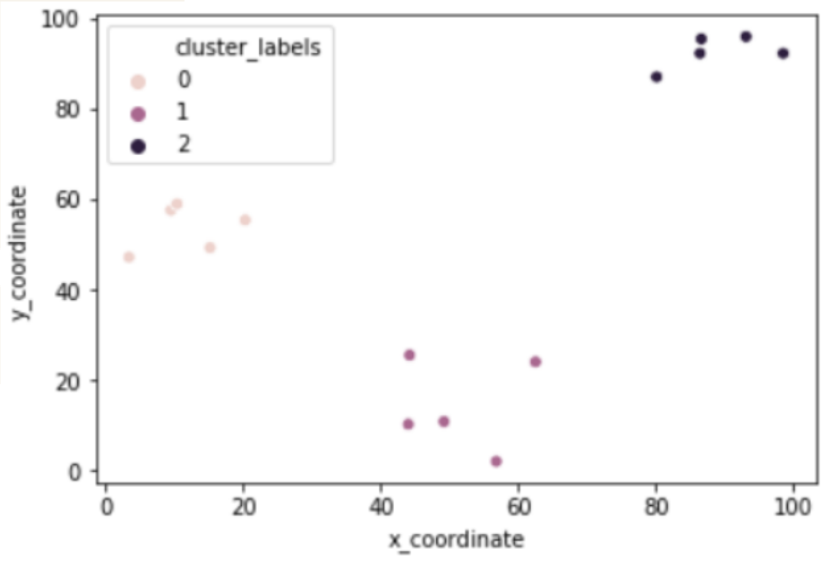
K-means clustering



```
from scipy.cluster.vq import kmeans, vq
from matplotlib import pyplot as plt
import seaborn as sns, pandas as pd

import random
random.seed((1000,2000))
```

```
x_coordinates = [80.1, 93.1, 86.6, 98.5, 86.4, 9.5, 15.2, 3.4,
                 10.4, 20.3, 44.2, 56.8, 49.2, 62.5, 44.0]
y_coordinates = [87.2, 96.1, 95.6, 92.4, 92.4, 57.7, 49.4,
                 47.3, 59.1, 55.5, 25.6, 2.1, 10.9, 24.1, 10.3]

df = pd.DataFrame({'x_coordinate': x_coordinates, 'y_coordinate': y_coordinates})
```

```
centroids,_ = kmeans(df, 3)
df['cluster_labels'], _ = vq(df, centroids)
```

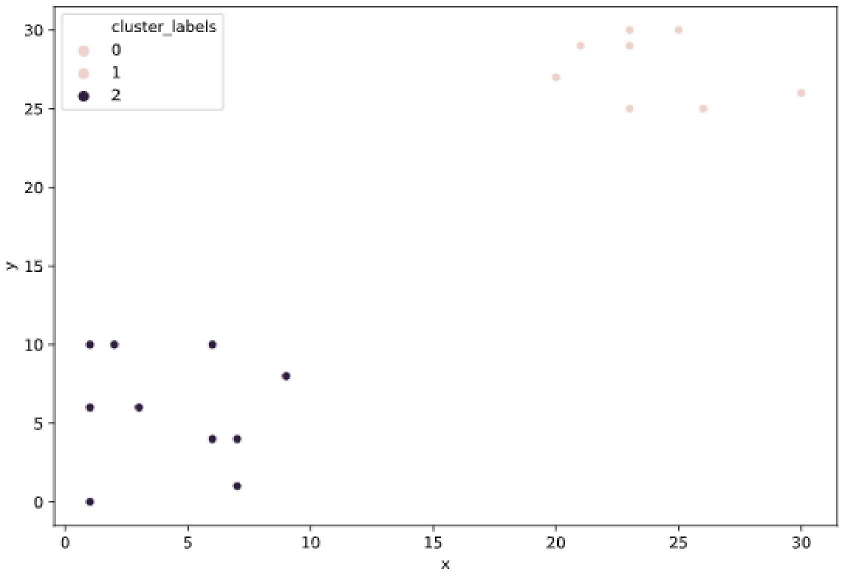


Pokémon sightings: hierarchical clustering

We are going to continue the investigation into the sightings of legendary Pokémon from the previous exercise. Remember that in the scatter plot of the previous exercise, you identified two areas where Pokémon sightings were dense. This means that the points seem to separate into two clusters. In this exercise, you will form two clusters of the sightings using hierarchical clustering.

`'x'` and `'y'` are columns of X and Y coordinates of the locations of sightings, stored in a Pandas data frame, `df`. The following are available for use: `matplotlib.pyplot` as `plt`, `seaborn` as `sns`, and `pandas` as `pd`.

```python
# Import linkage and fcluster functions
from scipy.cluster.hierarchy import linkage, fcluster

# Use the linkage() function to compute distances
Z = linkage(df, 'ward')

# Generate cluster labels
df['cluster_labels'] = fcluster(Z, 2, criterion='maxclust')

# Plot the points with seaborn
sns.scatterplot(x='x', y='y', hue='cluster_labels', data=df)
plt.show()
```



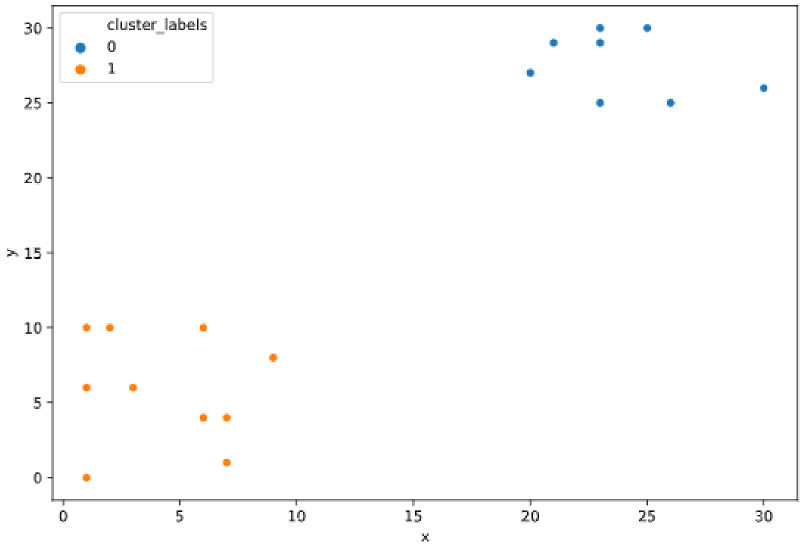
Notice that the cluster labels are plotted with different colors. You will notice that the resulting plot has an extra cluster labelled 0 in the legend. This will be explained later in the course.

Pokémon sightings: k-means clustering

We are going to continue the investigation into the sightings of legendary Pokémon from the previous exercise. Just like the previous exercise, we will use the same example of Pokémon sightings. In this exercise, you will form clusters of the sightings using k-means clustering.

`x` and `y` are columns of X and Y coordinates of the locations of sightings, stored in a Pandas data frame, `df`. The following are available for use: `matplotlib.pyplot` as `plt`, `seaborn` as `sns`, and `pandas` as `pd`.

```python
# Import kmeans and vq functions
from scipy.cluster.vq import kmeans, vq

# Compute cluster centers
centroids,_ = kmeans(df, 2)

# Assign cluster labels
df['cluster_labels'], _ = vq(df, centroids)

# Plot the points with seaborn
sns.scatterplot(x='x', y='y', hue='cluster_labels', data=df)
plt.show()
```



Notice that in this case, the results of both types of clustering are similar. We will look at distinctly different results later in the course.

Data preparation for cluster analysis

Variables have incomparable units (eg. product dimension in cm, price in $)
Variables with same units, but different scales and variances (eg. expenditures on cereals vs travel)

If we use data in this raw form, the results of clustering may be biased. The clusters formed may be dependent on 1 variable significantly more than the other.

Solution: normalization of data = process of rescaling data to a standard deviation of 1.

```python
from scipy.cluster.vq import whiten
```

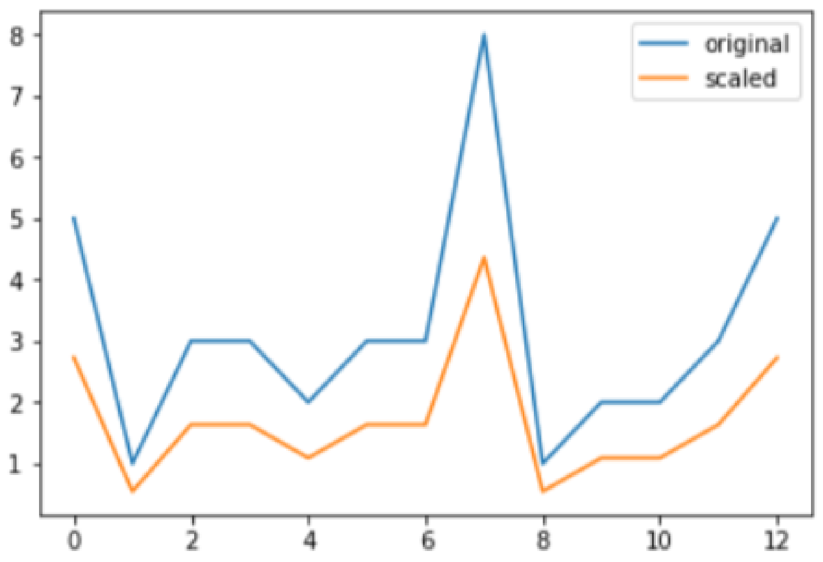
```python
data = [5, 1, 3, 3, 2, 3, 3, 8, 1, 2, 2, 3, 5]
```

```python
scaled_data = whiten(data)
print(scaled_data)
```

```
[2.73, 0.55, 1.64, 1.64, 1.09, 1.64, 1.64, 4.36, 0.55, 1.09, 1.09, 1.64, 2.73]
```

```
# Import plotting library
from matplotlib import pyplot as plt

# Initialize original, scaled data
plt.plot(data,
         label="original")
plt.plot(scaled_data,
         label="scaled")
```

```
# Show legend and display plot
plt.legend()
plt.show()
```



Variation in the scaled data has been toned down from the original data, but the trends remain the same.

Normalize basic list data

Now that you are aware of normalization, let us try to normalize some data. `goals_for` is a list of goals scored by a football team in their last ten matches. Let us standardize the data using the `whiten()` function.

```
# Import the whiten function
from scipy.cluster.vq import whiten

goals_for = [4,3,2,3,1,1,2,0,1,4]

# Use the whiten() function to standardize the data
scaled_data = whiten(goals_for)
print(scaled_data)
```

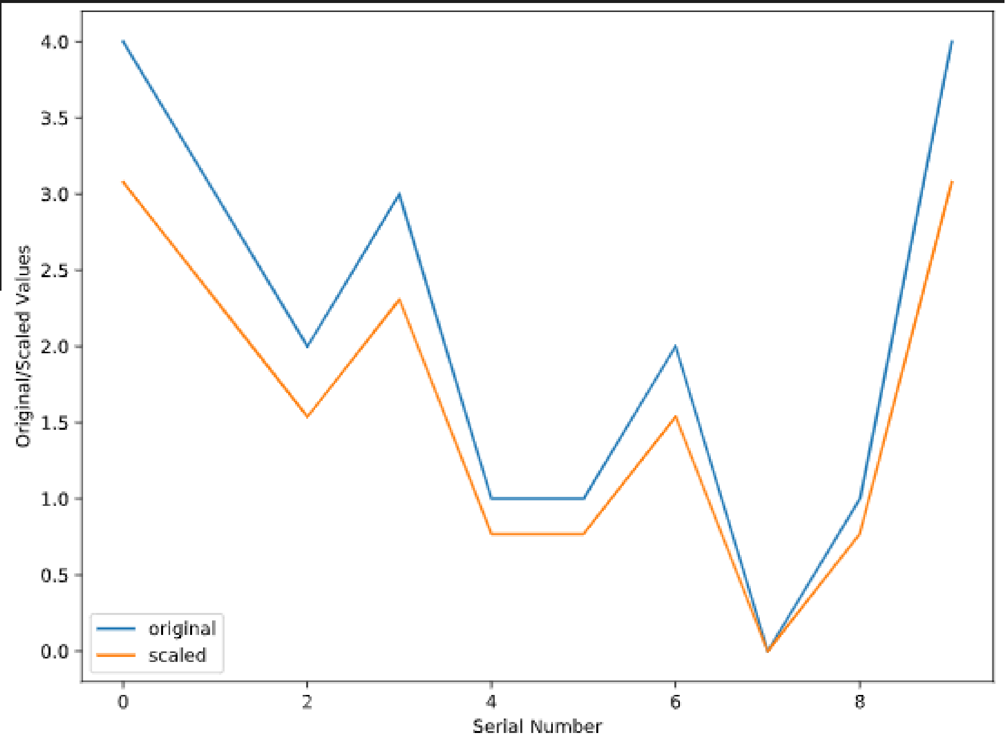
```
<script.py> output:
    [3.07692308 2.30769231 1.53846154 2.30769231 0.76923077 0.76923077
     1.53846154 0.         0.76923077 3.07692308]
```

Notice the scaled values have less variations in them. You will now visualize the data in the next exercise.

Visualize normalized data

After normalizing your data, you can compare the scaled data to the original data to see the difference. The variables from the last exercise, `goals_for` and `scaled_data` are already available to you.

```
# Plot original data
plt.plot(goals_for, label='original')

# Plot scaled data
plt.plot(scaled_data, label='scaled')

# Show the legend in the plot
plt.legend()

# Display the plot
plt.show()
```

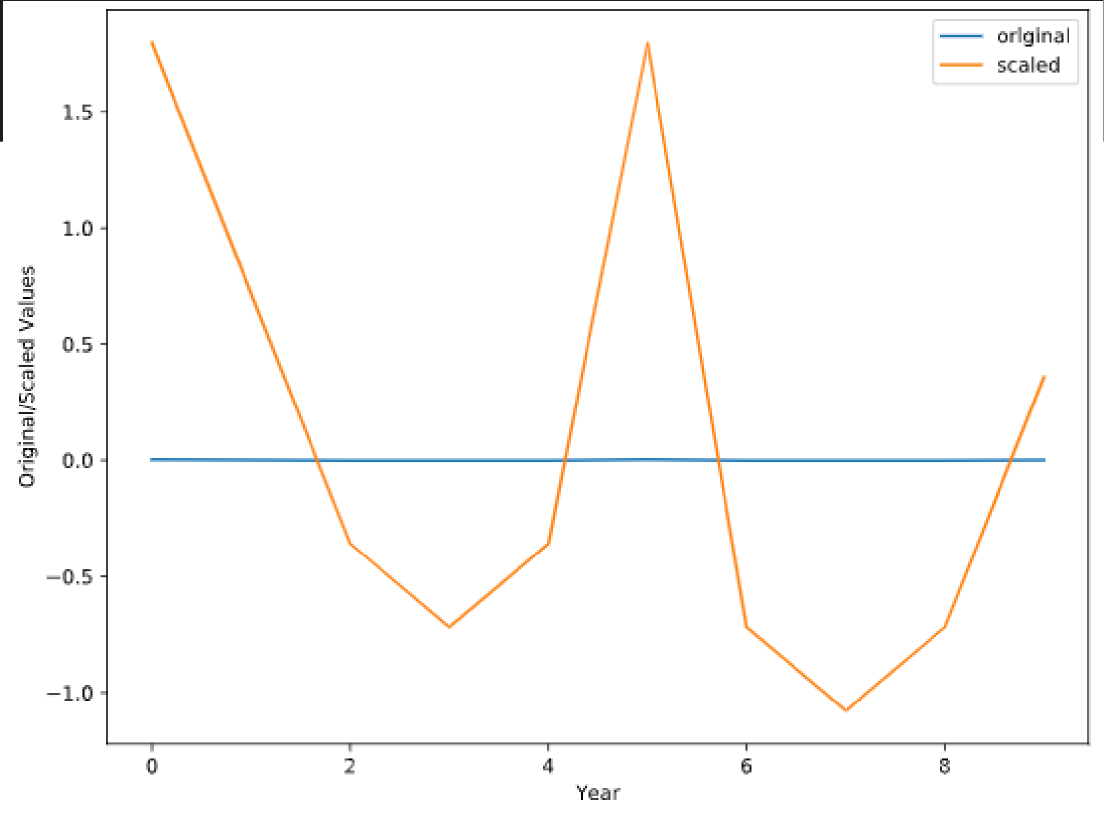


Notice the scaled values have lower variations in them.

Normalization of small numbers

In earlier examples, you have normalization of whole numbers. In this exercise, you will look at the treatment of fractional numbers - the change of interest rates in the country of Bangalla over the years. For your use, `matplotlib.pyplot` is imported as `plt`.

```
# Prepare data
rate_cuts = [0.0025, 0.001, -0.0005, -0.001, -0.0005, 0.0025, -0.001, -0.0015, -0.001, 0.0005]

# Use the whiten() function to standardize the data
scaled_data = whiten(rate_cuts)

# Plot original data
plt.plot(rate_cuts, label='original')

# Plot scaled data
plt.plot(scaled_data, label='scaled')

plt.legend()
plt.show()
```



Notice how the changes in the original data are negligible as compared to the scaled data.

FIFA 18: Normalize data

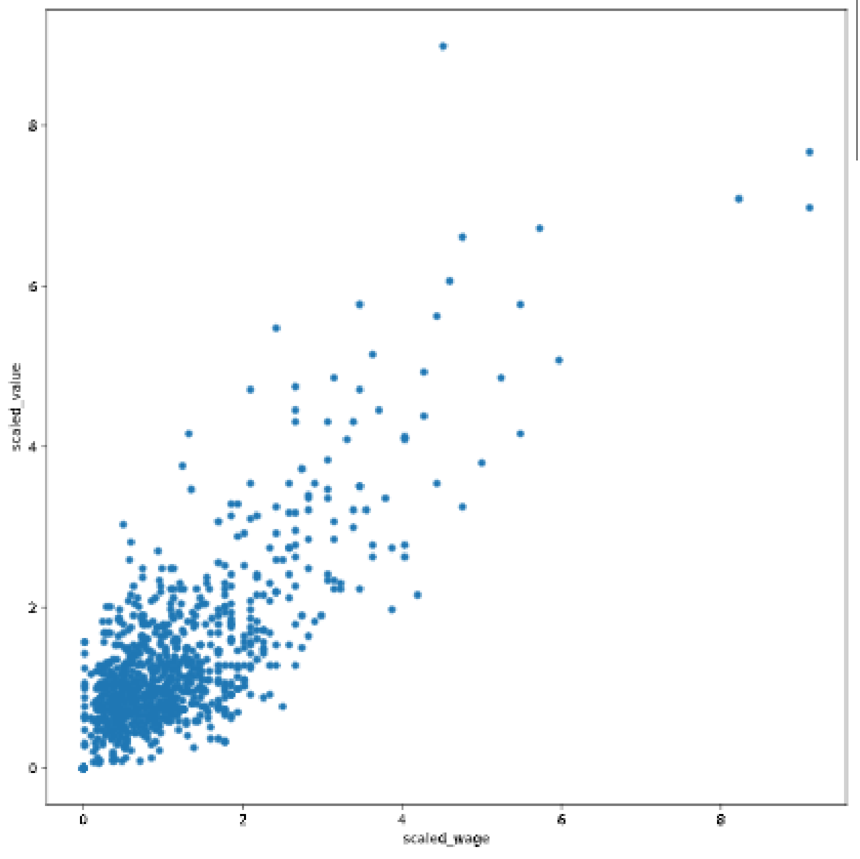
FIFA 18 is a football video game that was released in 2017 for PC and consoles. The dataset that you are about to work on contains data on the 1000 top individual players in the game. You will explore various features of the data as we move ahead in the course. In this exercise, you will work with two columns, `eur_wage`, the wage of a player in Euros and `eur_value`, their current transfer market value.

The data for this exercise is stored in a Pandas dataframe, `fifa`. `whiten` from `scipy.cluster.vq` and `matplotlib.pyplot` as `plt` have been pre-loaded.

```
# Scale wage and value
fifa['scaled_wage'] = whiten(fifa['eur_wage'])
fifa['scaled_value'] = whiten(fifa['eur_value'])

# Plot the two columns in a scatter plot
fifa.plot(x='scaled_wage', y='scaled_value', kind = 'scatter')
plt.show()

# Check mean and standard deviation of scaled values
print(fifa[['scaled_wage', 'scaled_value']].describe())
```

```
<script.py> output:
           scaled_wage  scaled_value
    count      1000.00       1000.00
    mean          1.12          1.31
    std           1.00          1.00
    min           0.00          0.00
    25%           0.47          0.73
    50%           0.85          1.02
    75%           1.41          1.54
    max           9.11          8.98
```

As you can see the scaled values have a standard deviation of 1.



Chapter 2. Hierarchical Clustering

This chapter focuses on a popular clustering algorithm - hierarchical clustering - and its implementation in SciPy. In addition to the procedure to perform hierarchical clustering, it attempts to help you answer an important question - how many clusters are present in your data? The chapter concludes with a discussion on the limitations of hierarchical clustering and discusses considerations while using hierarchical clustering.

Basics of hierarchical clustering

Creating a distance matrix using linkage (SciPy)

```
scipy.cluster.hierarchy.linkage(observations,
                                method='single',
                                metric='euclidean',
                                optimal_ordering=False
)
```

- `method` : how to calculate the proximity of clusters
- `metric` : distance metric
- `optimal_ordering` : order data points

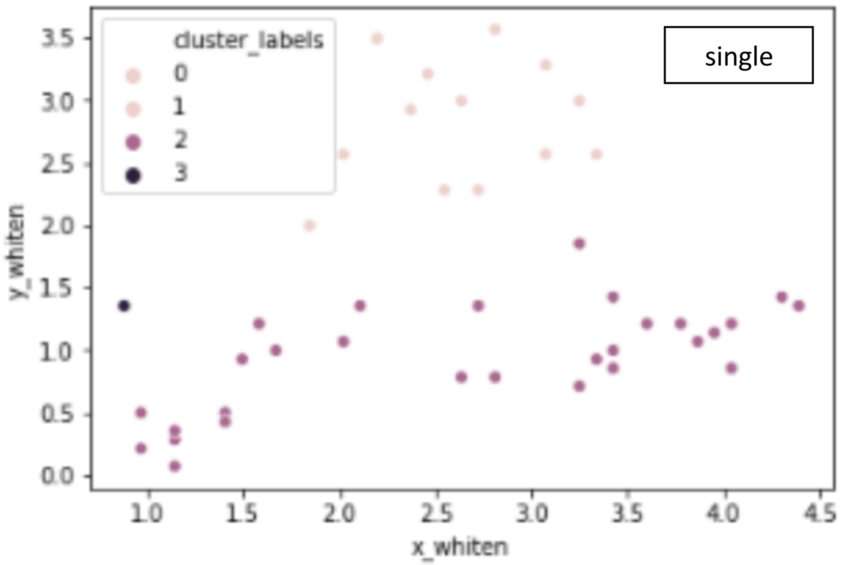
Creating cluster labels with fcluster

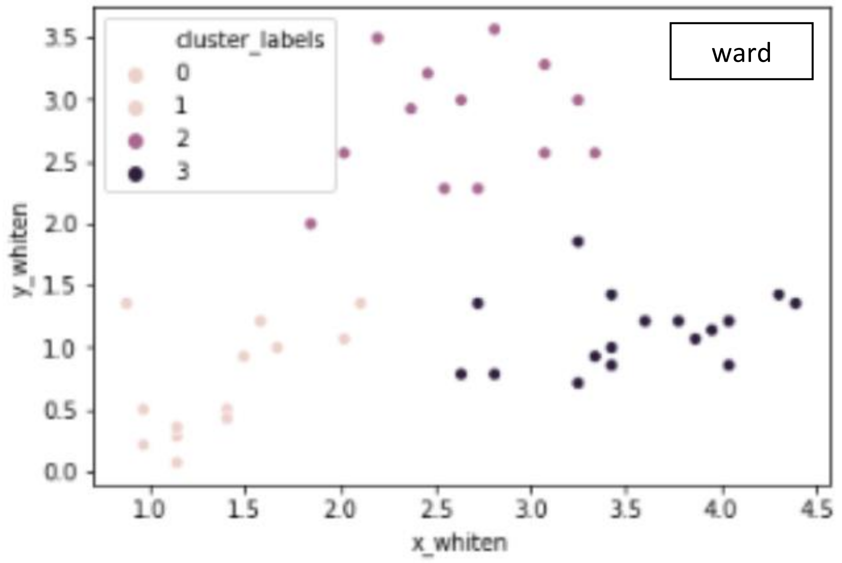
```
scipy.cluster.hierarchy.fcluster(distance_matrix,
                                 num_clusters,
                                 criterion
)
```

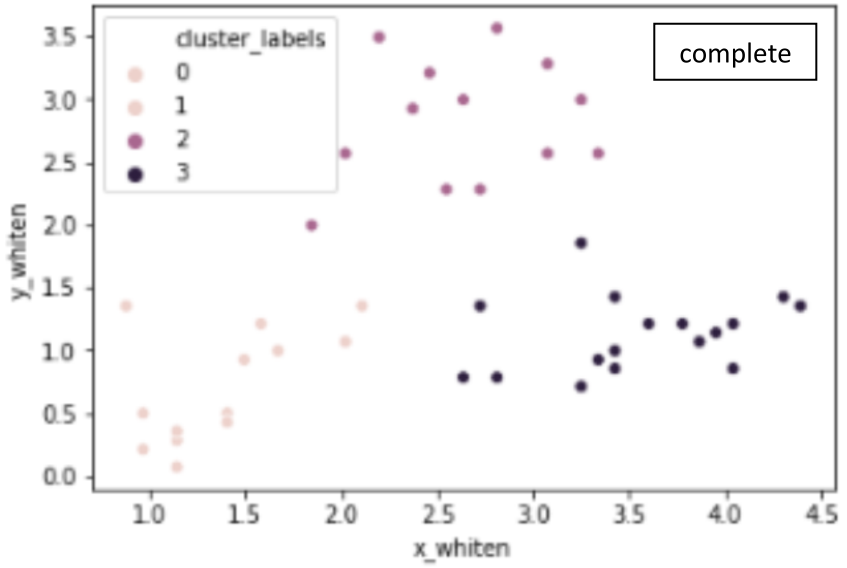
- `distance_matrix` : output of `linkage()` method
- `num_clusters` : number of clusters
- `criterion` : how to decide thresholds to form clusters

Which method should use?

- single: based on two closest objects
- complete: based on two farthest objects
- average: based on the arithmetic mean of all objects
- centroid: based on the geometric mean of all objects
- median: based on the median of all objects
- ward: based on the sum of squares







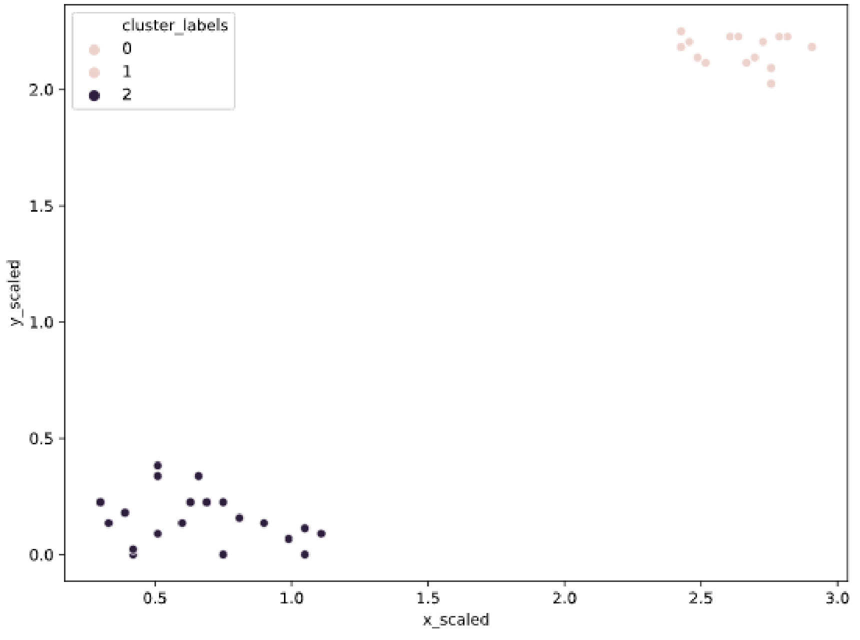
There is no right method that could be applied to all problems. Need to carefully study the data, understand the distribution, and decide which method is right for the data.

Hierarchical clustering: ward method

It is time for Comic-Con! Comic-Con is an annual comic-based convention held in major cities in the world. You have the data of last year's footfall, the number of people at the convention ground at a given time. You would like to decide the location of your stall to maximize sales. Using the ward method, apply hierarchical clustering to find the two points of attraction in the area.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time.

```python
# Import the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster, linkage

# Use the linkage() function
distance_matrix = linkage(comic_con[['x_scaled', 'y_scaled']], method = 'ward', metric = 'euclidean')

# Assign cluster labels
comic_con['cluster_labels'] = fcluster(distance_matrix, 2, criterion='maxclust')

# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled', 
                hue='cluster_labels', data = comic_con)
plt.show()
```



Notice the two clusters correspond to the points of attractions in the figure towards the bottom (a stage) and the top right (an interesting stall).

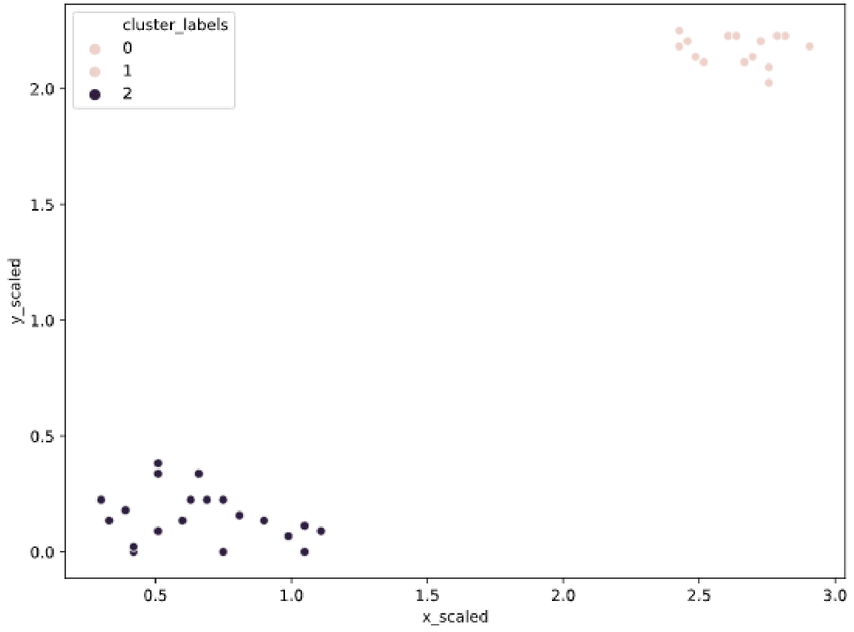
Hierarchical clustering: single method

Let us use the same footfall dataset and check if any changes are seen if we use a different method for clustering.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time.

```
# Import the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster, linkage

# Use the linkage() function
distance_matrix = linkage(comic_con[['x_scaled', 'y_scaled']], method = 'single', metric = 'euclidean')

# Assign cluster labels
comic_con['cluster_labels'] = fcluster(distance_matrix, 2, criterion='maxclust')

# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = comic_con)
plt.show()
```

Notice that in this example, the clusters formed are not different from the ones created using the ward method.



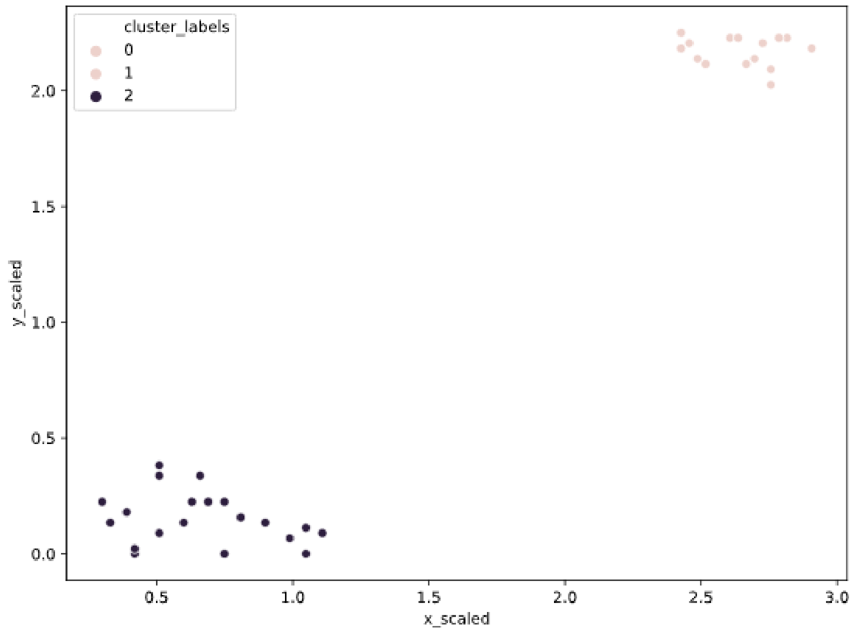
Hierarchical clustering: complete method

For the third and final time, let us use the same footfall dataset and check if any changes are seen if we use a different method for clustering.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time.

```
# Import the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster, linkage

# Use the linkage() function
distance_matrix = linkage(comic_con[['x_scaled', 'y_scaled']], method = 'complete', metric = 'euclidean')

# Assign cluster labels
comic_con['cluster_labels'] = fcluster(distance_matrix, 2, criterion='maxclust')

# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = comic_con)
plt.show()
```

Coincidentally, the clusters formed are not different from the ward or single methods. Next, let us learn how to visualize clusters.



Visualize clusters

Why visualise clusters? To make sense of the clusters formed, validation of the clusters, spot trends in data.

Using matplotlib

```python
from matplotlib import pyplot as plt
```

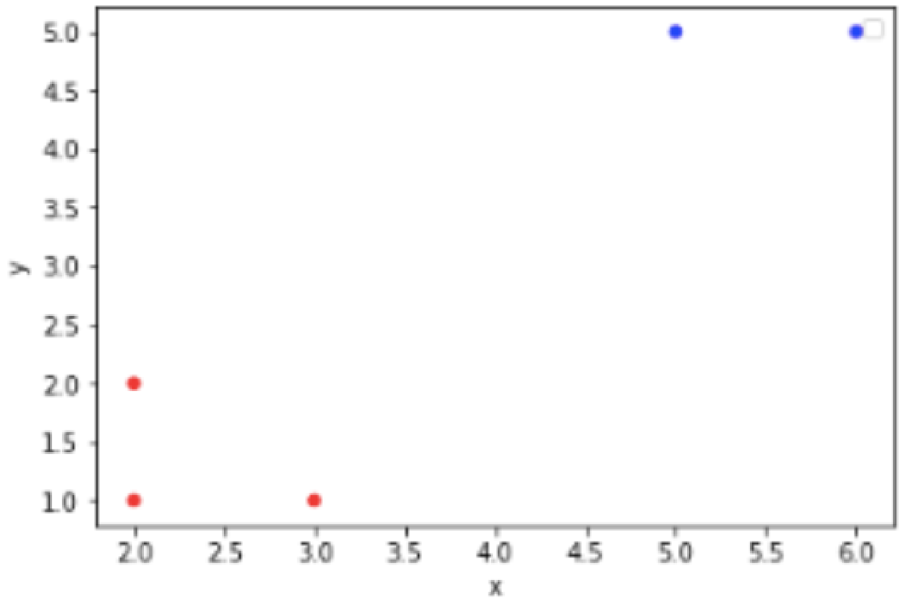
```python
df = pd.DataFrame({'x': [2, 3, 5, 6, 2],
                   'y': [1, 1, 5, 5, 2],
                   'labels': ['A', 'A', 'B', 'B', 'A']})
colors = {'A':'red', 'B':'blue'}
df.plot.scatter(x='x',
                y='y',
                c=df['labels'].apply(lambda x: colors[x]))
plt.show()
```

Using seaborn (preferred)

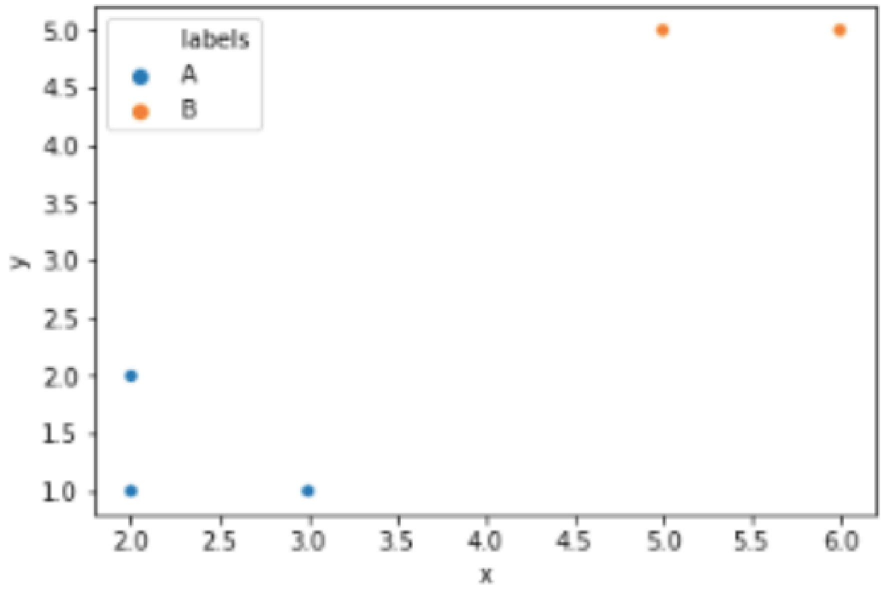
```python
from matplotlib import pyplot as plt
import seaborn as sns
```

```python
df = pd.DataFrame({'x': [2, 3, 5, 6, 2],
                   'y': [1, 1, 5, 5, 2],
                   'labels': ['A', 'A', 'B', 'B', 'A']})
sns.scatterplot(x='x',
                y='y',
                hue='labels',
                data=df)
plt.show()
```

MATPLOTLIB PLOT



SEABORN PLOT



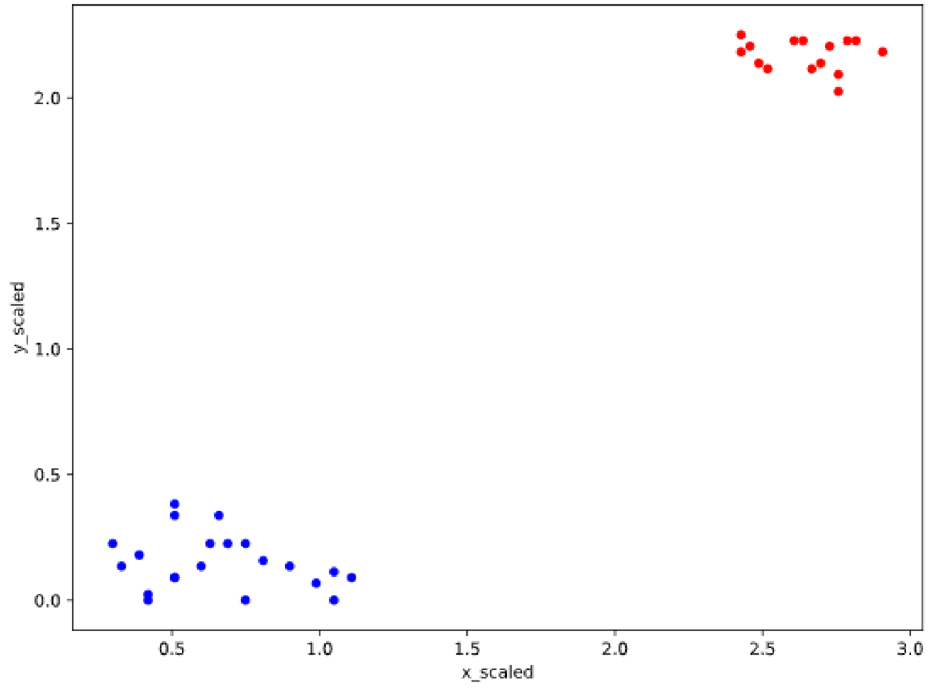
Visualize clusters with matplotlib

We have discussed that visualizations are necessary to assess the clusters that are formed and spot trends in your data. Let us now focus on visualizing the footfall dataset from Comic-Con using the `matplotlib` module.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time. `cluster_labels` has the cluster labels. A linkage object is stored in the variable `distance_matrix`.

```
# Import the pyplot class
from matplotlib import pyplot as plt

# Define a colors dictionary for clusters
colors = {1:'red', 2:'blue'}

# Plot a scatter plot
comic_con.plot.scatter(x='x_scaled',
                       y='y_scaled',
                       c=comic_con['cluster_labels'].apply(lambda x: colors[x]))
plt.show()
```

The two different clusters are shown in different colors.

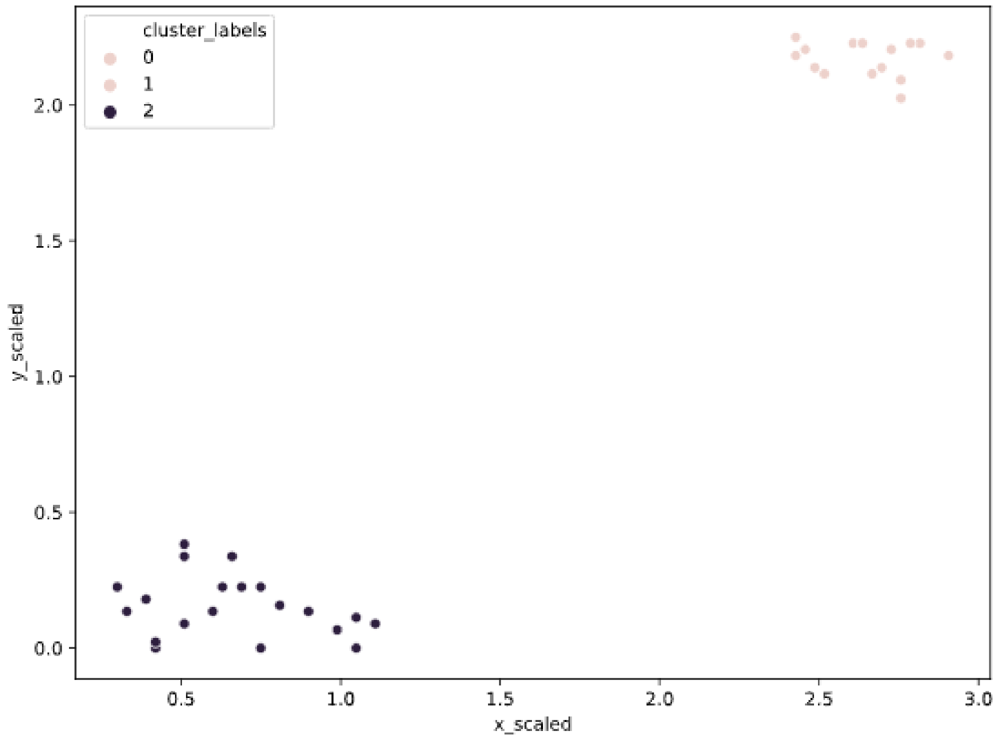


Visualize clusters with seaborn

Let us now visualize the footfall dataset from Comic Con using the `seaborn` module. Visualizing clusters using `seaborn` is easier with the inbuild `hue` function for cluster labels.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time. `cluster_labels` has the cluster labels. A linkage object is stored in the variable `distance_matrix`.

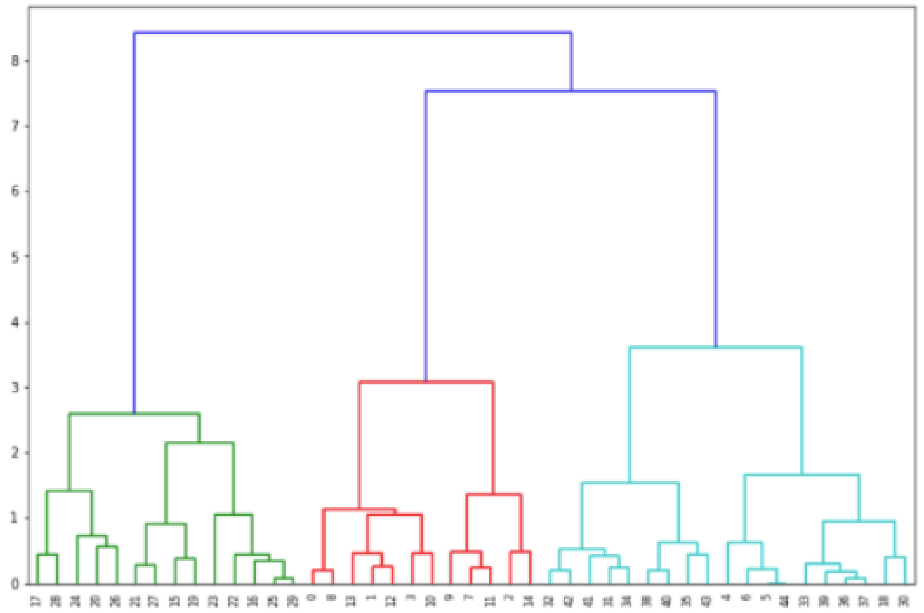
```
# Import the seaborn module
import seaborn as sns

# Plot a scatter plot using seaborn
sns.scatterplot(x='x_scaled',
                y='y_scaled',
                hue='cluster_labels',
                data=comic_con)
plt.show()
```



Notice the legend is automatically shown when using the hue argument.

How many clusters?

Basic strategy is to decide clusters on visual inspection.



A better way is to decide clusters using dendrograms, by showing progressions as clusters are merged.

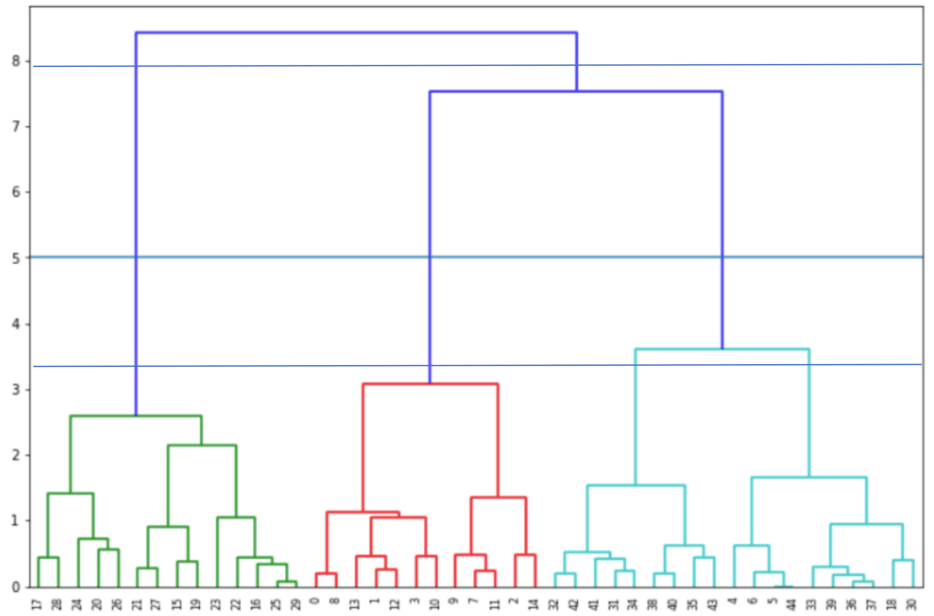
Dendrogram = a branching diagram that demonstrates the progression in a linkage object as we proceed through the hierarchical clustering algorithm, and how each cluster is conposed by branching out into its child nodes.

The x-axis represent individual points, and the y-axis represents the distance or dissimilarity between clusters.

Creating a dendrogram using **SciPy**:

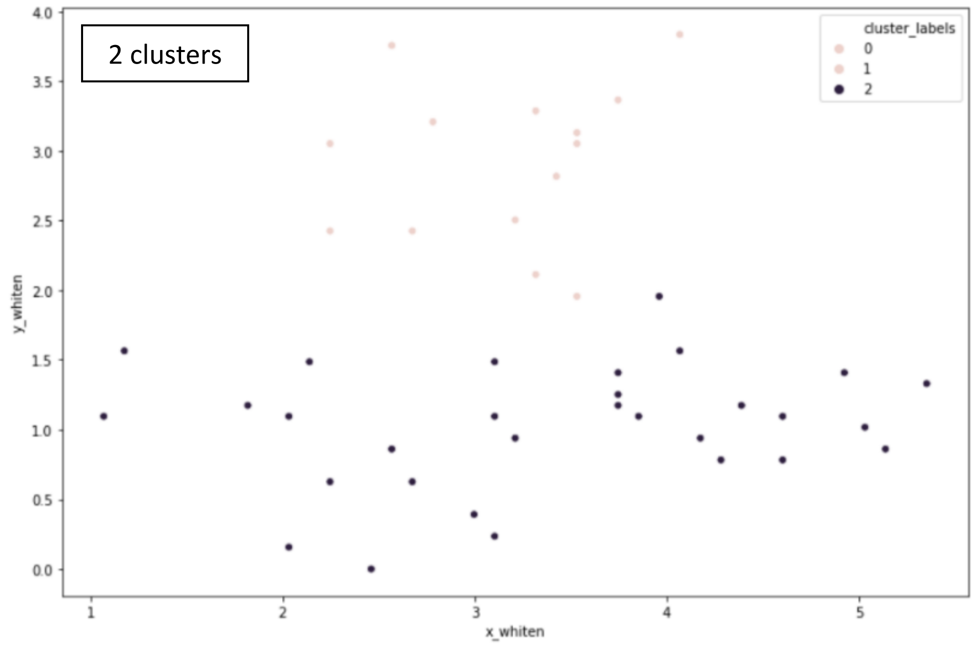
```
from scipy.cluster.hierarchy import dendrogram
```

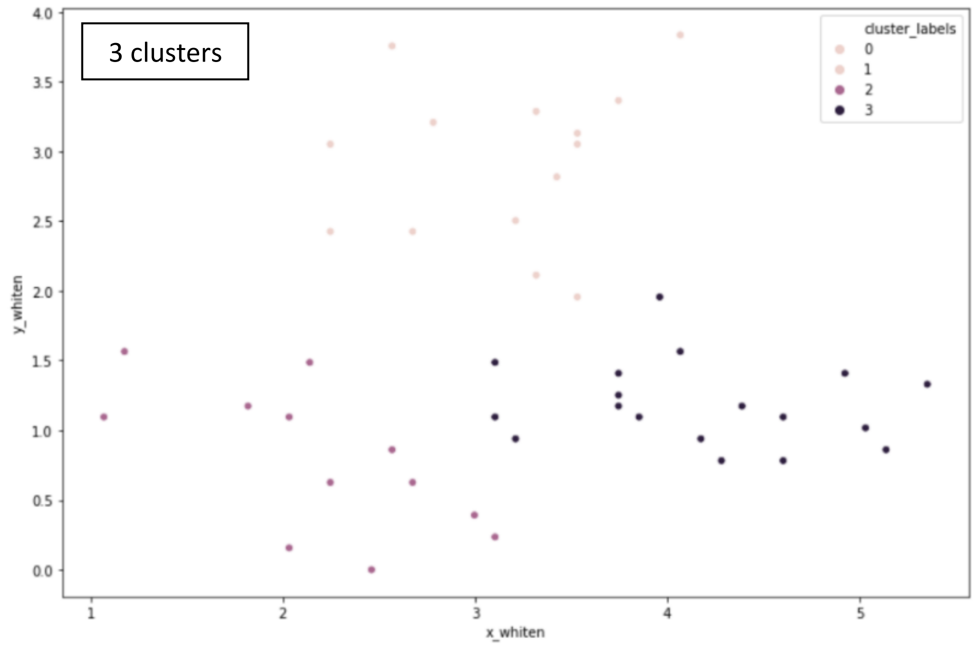
```
Z = linkage(df[['x_whiten', 'y_whiten']],
            method='ward',
            metric='euclidean')
dn = dendrogram(Z)
plt.show()
```

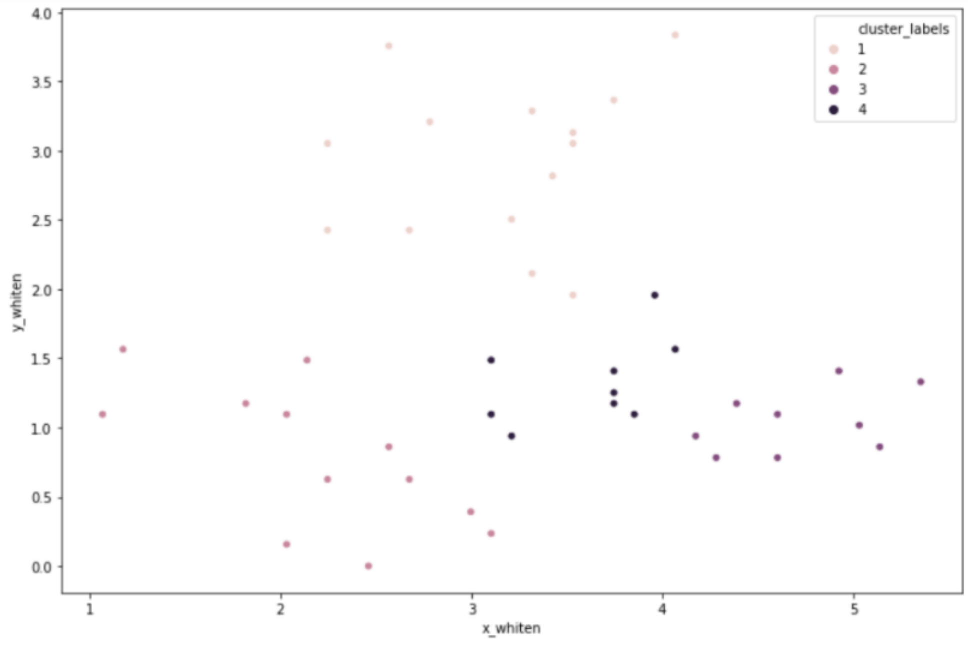


Horizontal lines cuts through n number of clusters. In the example on the right, the horizontal line cuts through 2, 3, or 4 clusters.

An additional check of visualising the data may be performed before deciding on the number of clusters.







Looking at these plots, 2 or 3 or 4 clusters all make sense.

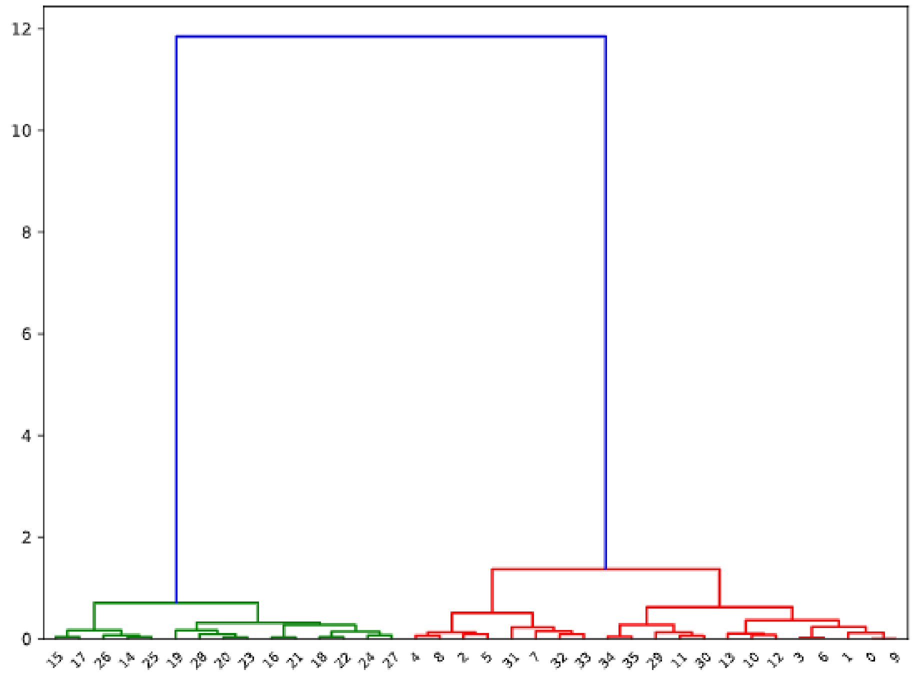
So we could use distance metric and allowable thresholds to help make the decision.

Create a dendrogram

Dendrograms are branching diagrams that show the merging of clusters as we move through the distance matrix. Let us use the Comic Con footfall data to create a dendrogram.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time. `cluster_labels` has the cluster labels. A linkage object is stored in the variable `distance_matrix`.

```python
# Import the dendrogram function
from scipy.cluster.hierarchy import dendrogram

# Create a dendrogram
dn = dendrogram(distance_matrix)

# Display the dendogram
plt.show()
```



Notice the significant difference between the inter-cluster distances beyond the top two clusters.

How many clusters in comic con data?

Given the dendrogram from the last exercise, how many clusters can you see in the data?

A dendrogram is stored in the variable `dn`. Use `plt.show()` to display the dendrogram.

Possible Answers

- [✓ 2 clusters]
- 3 clusters
- 4 clusters

Answer: Notice that the top two clusters are farthest away from each other.

Limitations of hierarchical clustering

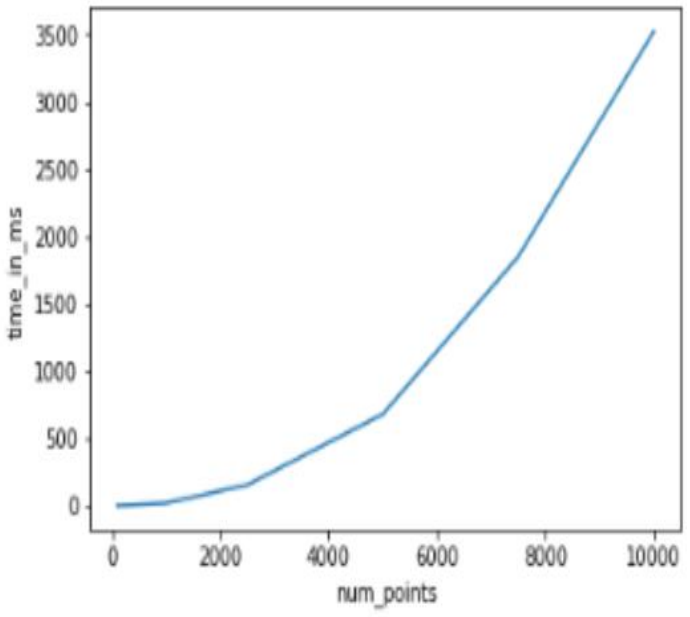
To measuring the speed in hierarchical clustering, by measuring the speed of `.linkage()` method on randonly generated points.

```python
from scipy.cluster.hierarchy import linkage
import pandas as pd
import random, timeit
points = 100
df = pd.DataFrame({'x': random.sample(range(0, points), points),
                   'y': random.sample(range(0, points), points)})
%timeit linkage(df[['x', 'y']], method = 'ward', metric = 'euclidean')
```

```
1.02 ms ± 133 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The linkage method has a quadratic increase in runtime with increased number of data points.

This makes the technique of hierarchical clustering infeasible for large datasets, for example shopping habits of all customers in Walmart in a year.



Timing run of hierarchical clustering

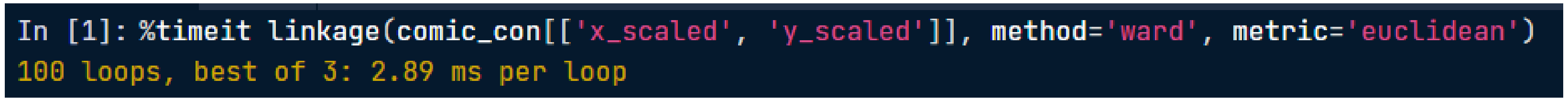
In earlier exercises of this chapter, you have used the data of Comic-Con footfall to create clusters. In this exercise you will time how long it takes to run the algorithm on DataCamp's system.

Remember that you can time the execution of small code snippets with:

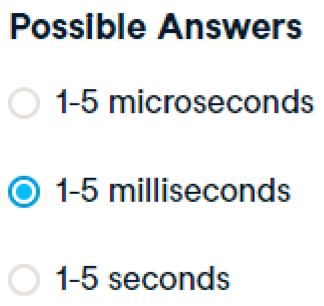
```
%timeit sum([1, 3, 2])
```

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time. The `timeit` module and `linkage` function are already imported.

How long does it take to the run the `linkage` function on the comic con data?



```
In [1]: %timeit linkage(comic_con[['x_scaled', 'y_scaled']], method='ward', metric='euclidean')
100 loops, best of 3: 2.89 ms per loop
```



Possible Answers

- 1-5 microseconds
- [✓ 1-5 milliseconds]
- 1-5 seconds

It took only a few milliseconds to run the data.

FIFA 18: exploring defenders

In the FIFA 18 dataset, various attributes of players are present. Two such attributes are:

- *sliding tackle*: a number between 0-99 which signifies how accurate a player is able to perform sliding tackles
- *aggression*: a number between 0-99 which signifies the commitment and will of a player

These are typically high in defense-minded players. In this exercise, you will perform clustering based on these attributes in the data.

This data consists of 5000 rows, and is considerably larger than earlier datasets. Running hierarchical clustering on this data can take up to 10 seconds.

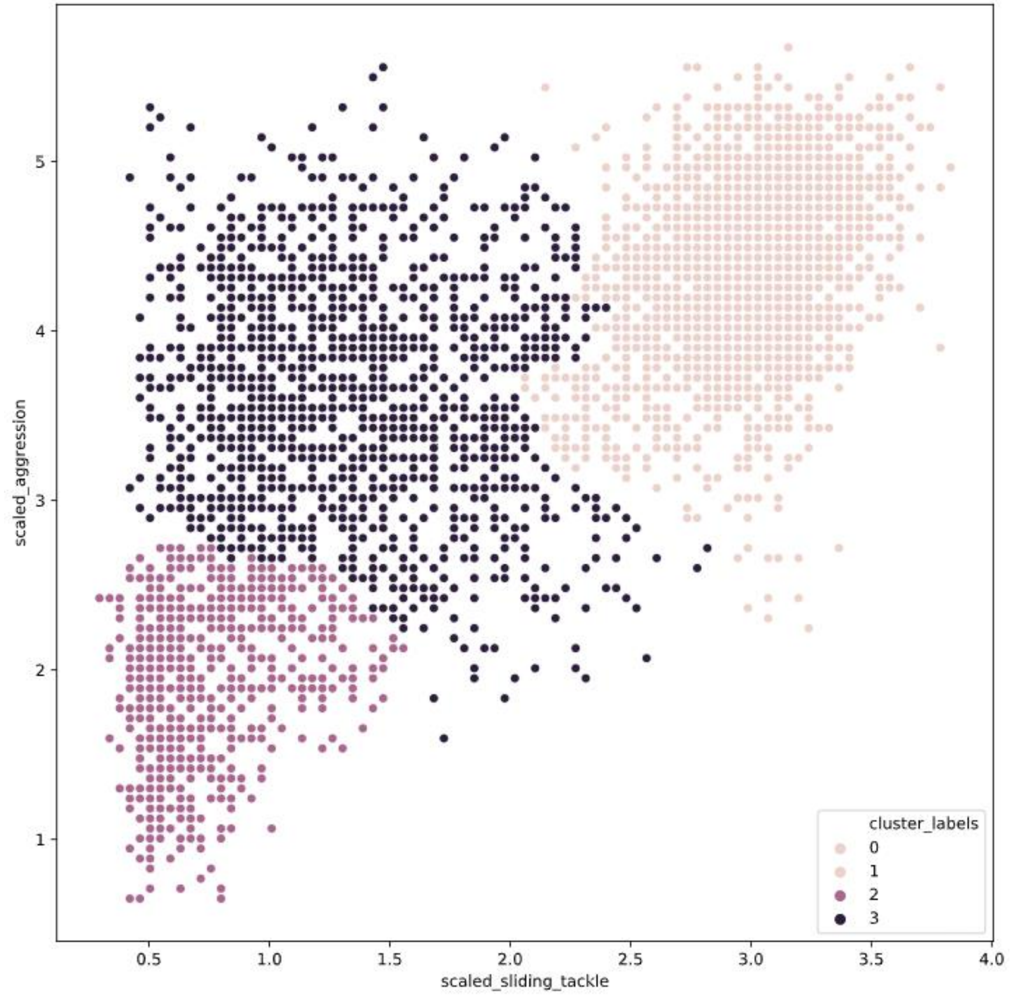
The following modules are pre-loaded: `dendrogram`, `linkage`, `fcluster` from `scipy.cluster.hierarchy`, `matplotlib.pyplot` as `plt`, `seaborn` as `sns`. The data is stored in a Pandas dataframe, `fifa`.

```python
# Fit the data into a hierarchical clustering algorithm
distance_matrix = linkage(fifa[['scaled_sliding_tackle', 'scaled_aggression']], 'ward')

# Assign cluster labels to each row of data
fifa['cluster_labels'] = fcluster(distance_matrix, 3, criterion='maxclust')

# Display cluster centers of each cluster
print(fifa[['scaled_sliding_tackle', 'scaled_aggression', 'cluster_labels']].groupby('cluster_labels').mean())

# Create a scatter plot through seaborn
sns.scatterplot(x='scaled_sliding_tackle', y='scaled_aggression', hue='cluster_labels', data=fifa)
plt.show()
```

```
<script.py> output:
                    scaled_sliding_tackle  scaled_aggression
    cluster_labels
    1                                2.99               4.35
    2                                0.74               1.94
    3                                1.34               3.62
```

```
<script.py> output:
                    scaled_sliding_tackle  scaled_aggression
    cluster_labels
    1                                2.99               4.35
    2                                0.74               1.94
    3                                1.34               3.62
```

Notice how long it took to run hierarchical clustering on a few thousand data points.
In the next chapter, you will explore clusters in data through k-means clustering.



Chapter 3. K–Means Clustering

This chapter introduces a different clustering algorithm - k-means clustering - and its implementation in SciPy. K-means clustering overcomes the biggest drawback of hierarchical clustering that was discussed in the last chapter. As dendrograms are specific to hierarchical clustering, this chapter discusses one method to find the number of clusters before running k-means clustering. The chapter concludes with a discussion on the limitations of k-means clustering and discusses considerations while using this algorithm.

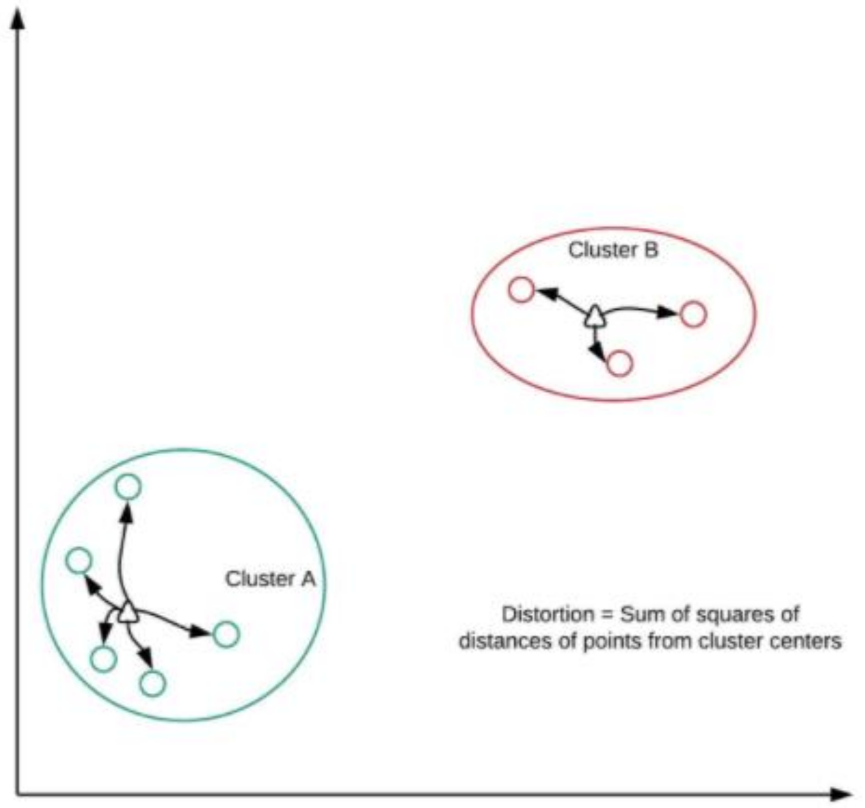
Basics of k-means clustering

Allows clustering large datasets in a fraction of the time compared to hierarchical clustering.

```
kmeans(obs, k_or_guess, iter, thresh, check_finite)
```

- `obs` : standardized observations
- `k_or_guess` : number of clusters
- `iter` : number of iterations (default: 20)
- `thres` : threshold (default: 1e-05)
- `check_finite` : whether to check if observations contain contain only finite numbers (default: True)

Returns two objects: cluster centers, distortion


Cluster B
Cluster A
Distortion = Sum of squares of distances of points from cluster centers

```
vq(obs, code_book, check_finite=True)
```

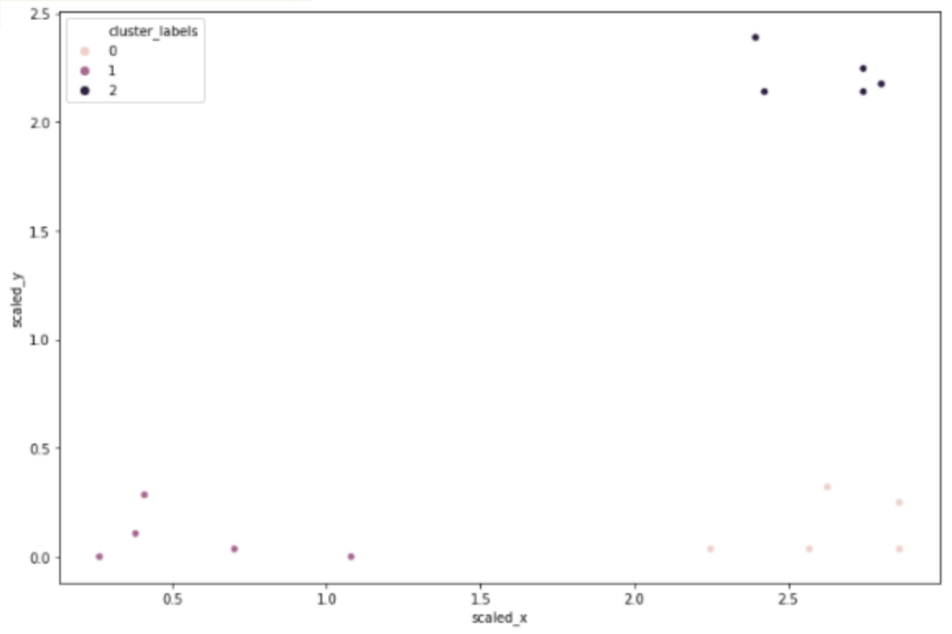
- `obs` : standardized observations
- `code_book` : cluster centers
- `check_finite` : whether to check if observations contain only finite numbers (default: True)

Returns two objects: a list of cluster labels, a list of distortions

```
# Import kmeans and vq functions
from scipy.cluster.vq import kmeans, vq
```

```
# Generate cluster centers and labels
cluster_centers, _ = kmeans(df[['scaled_x', 'scaled_y']], 3)
df['cluster_labels'], _ = vq(df[['scaled_x', 'scaled_y']], cluster_centers)
```

```
# Plot clusters
sns.scatterplot(x='scaled_x', y='scaled_y', hue='cluster_labels', data=df)
plt.show()
```



Notice there are 3 distinct clusters in the figure.

K-means clustering: first exercise

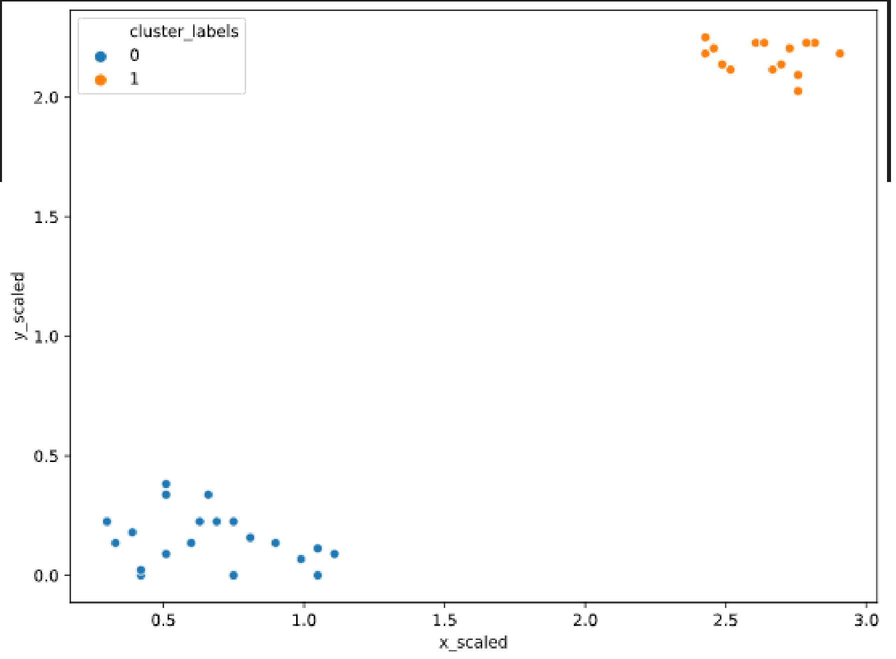
This exercise will familiarize you with the usage of k-means clustering on a dataset. Let us use the Comic Con dataset and check how k-means clustering works on it.

Recall the two steps of k-means clustering:

- Define cluster centers through `kmeans()` function. It has two required arguments: observations and number of clusters.
- Assign cluster labels through the `vq()` function. It has two required arguments: observations and cluster centers.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time.

```
# Import the kmeans and vq functions
from scipy.cluster.vq import kmeans, vq

# Generate cluster centers
cluster_centers, distortion = kmeans(comic_con[['x_scaled', 'y_scaled']], 2)

# Assign cluster labels
comic_con['cluster_labels'], distortion_list = vq(comic_con[['x_scaled', 'y_scaled']], cluster_centers)

# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = comic_con)
plt.show()
```



Notice that the clusters formed are exactly the same as hierarchical clustering that you did in the previous chapter.

Runtime of k-means clustering

Recall that it took a significantly long time to run hierarchical clustering. How long does it take to run the `kmeans()` function on the FIFA dataset?

The data is stored in a Pandas data frame, `fifa`. `scaled_sliding_tackle` and `scaled_aggression` are the relevant scaled columns. `timeit` and `kmeans` have been imported.

Cluster centers are defined through the `kmeans()` function. It has two required arguments: observations and number of clusters. You can use `%timeit` before a piece of code to check how long it takes to run. You can time the `kmeans()` function for three clusters on the `fifa` dataset.

```
In [1]: %timeit kmeans(fifa[['scaled_sliding_tackle', 'scaled_aggression']], 2)
10 loops, best of 3: 38.2 ms per loop
```

It took only about 5 seconds to run hierarchical clustering on this data, but only 50 milliseconds to run k-means clustering.

Possible Answers

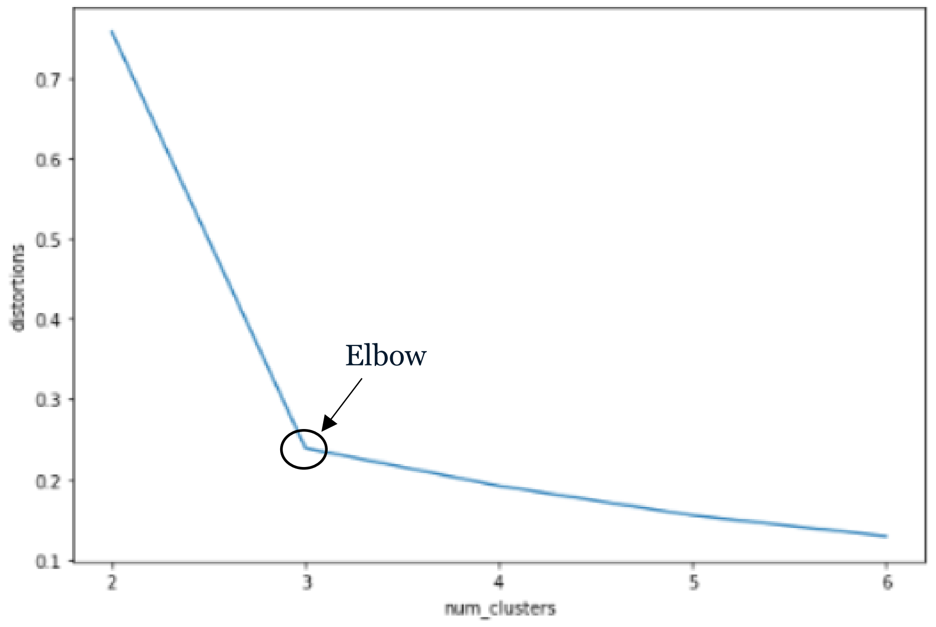
- ~ 50 µs (microseconds)
- [✓ ~ 50 ms (milliseconds)]
- ~ 50 s (seconds)

How many clusters?

In k-means clustering, there is no absolute way of finding out how many clusters exist in your dataset.

But there are certain indicative methods, including constructing an elbow plot to decide the right number of clusters for your dataset.

Elbow plot = distortion vs num_clusters



```
# Declaring variables for use
distortions = []

num_clusters = range(2, 7)
```

```
# Populating distortions for various clusters
for i in num_clusters:
    centroids, distortion = kmeans(df[['scaled_x', 'scaled_y']], i)
    distortions.append(distortion)
```

```
# Plotting elbow plot data
elbow_plot_data = pd.DataFrame({'num_clusters': num_clusters,
                                'distortions': distortions})

sns.lineplot(x='num_clusters', y='distortions',
             data = elbow_plot_data)
plt.show()
```

Distortion decreases sharply from 2 to 3 clusters, but has a very gradual decrease with a subsequent increase in number of clusters. Therefore the ideal numer of clusters is 3. This is only an indication.

There are other indicative methods: average silhouette and gap statistic.

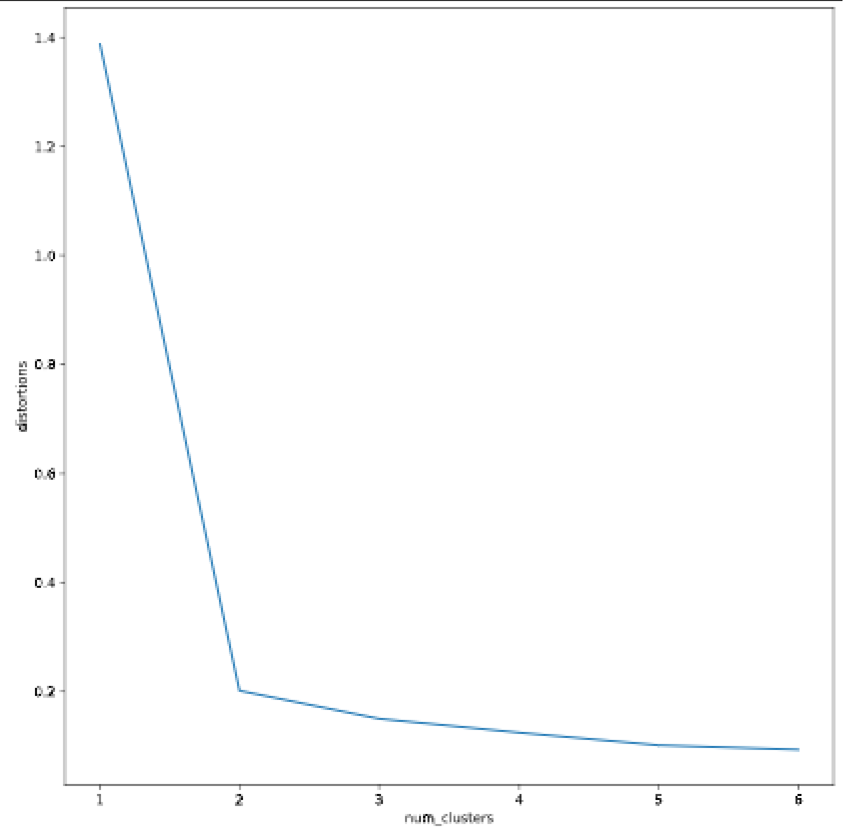
Elbow method on distinct clusters

Let us use the comic con data set to see how the elbow plot looks on a data set with distinct, well-defined clusters. You may want to display the data points before proceeding with the exercise.

The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time.

```
distortions = []
num_clusters = range(1, 7)
```

```
# Create a list of distortions from the kmeans function
for i in num_clusters:
    cluster_centers, distortion = kmeans(comic_con[['x_scaled', 'y_scaled']], i)
    distortions.append(distortion)

# Create a data frame with two lists - num_clusters, distortions
elbow_plot = pd.DataFrame({'num_clusters': num_clusters, 'distortions': distortions})

# Creat a line plot of num_clusters and distortions
sns.lineplot(x='num_clusters', y='distortions', data=elbow_plot)
plt.xticks(num_clusters)
plt.show()
```



Question:

From the elbow plot, how many clusters are there in the data?

Possible Answers

[✓ 2 clusters]

4 clusters

6 clusters

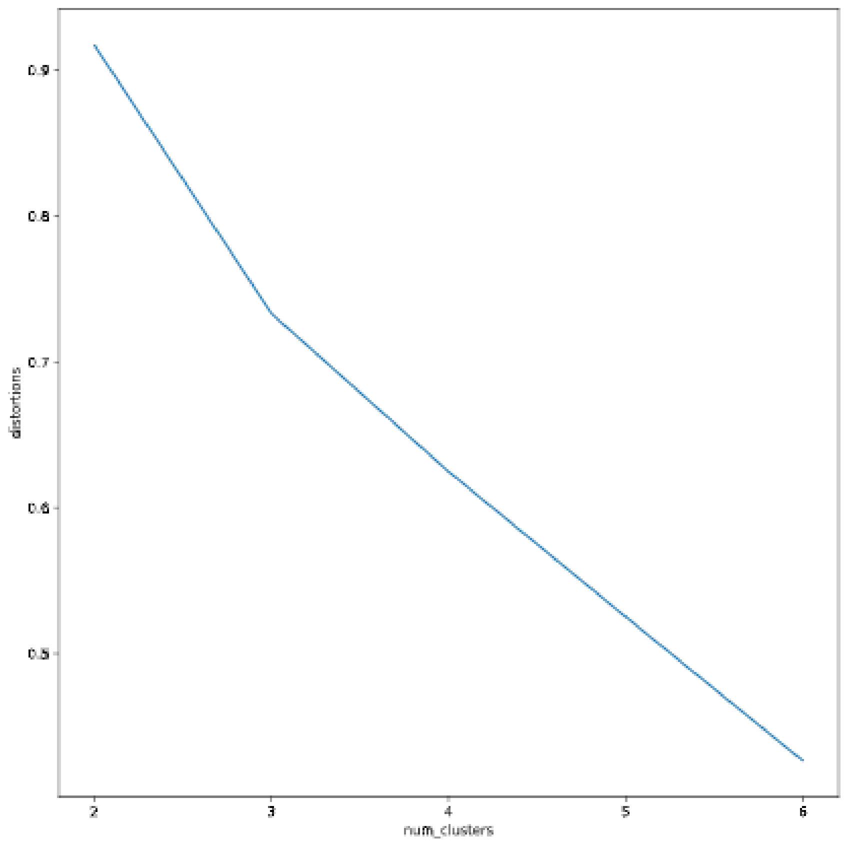
Answer: 2 clusters

Elbow method on uniform data

In the earlier exercise, you constructed an elbow plot on data with well-defined clusters. Let us now see how the elbow plot looks on a data set with uniformly distributed points. You may want to display the data points on the console before proceeding with the exercise.

The data is stored in a Pandas data frame, `uniform_data`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of points.

```python
distortions = []
num_clusters = range(2, 7)

# Create a list of distortions from the kmeans function
for i in num_clusters:
    cluster_centers, distortion = kmeans(uniform_data[['x_scaled', 'y_scaled']], i)
    distortions.append(distortion)

# Create a data frame with two lists - number of clusters and distortions
elbow_plot = pd.DataFrame({'num_clusters': num_clusters, 'distortions': distortions})

# Creat a line plot of num_clusters and distortions
sns.lineplot(x='num_clusters', y='distortions', data=elbow_plot)
plt.xticks(num_clusters)
plt.show()
```



Question:

From the elbow plot, how many clusters are there in the data?

Possible Answers

[✓ Can not be determined]

3 clusters

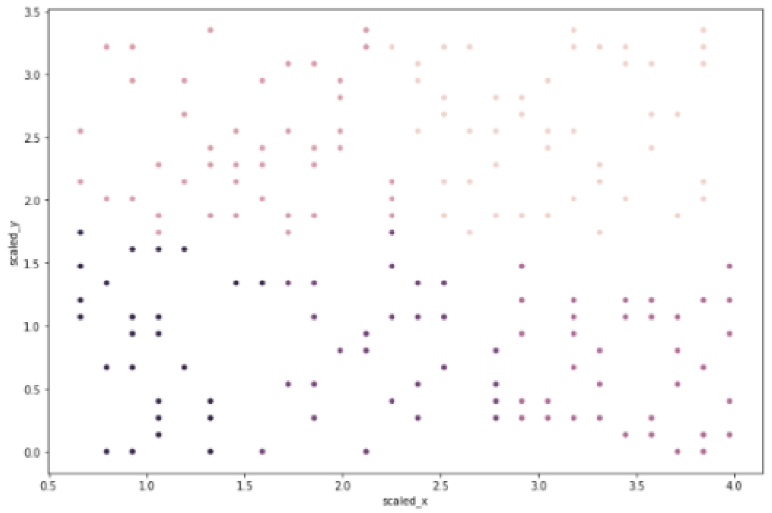
4 clusters

There is no well defined elbow in this plot!

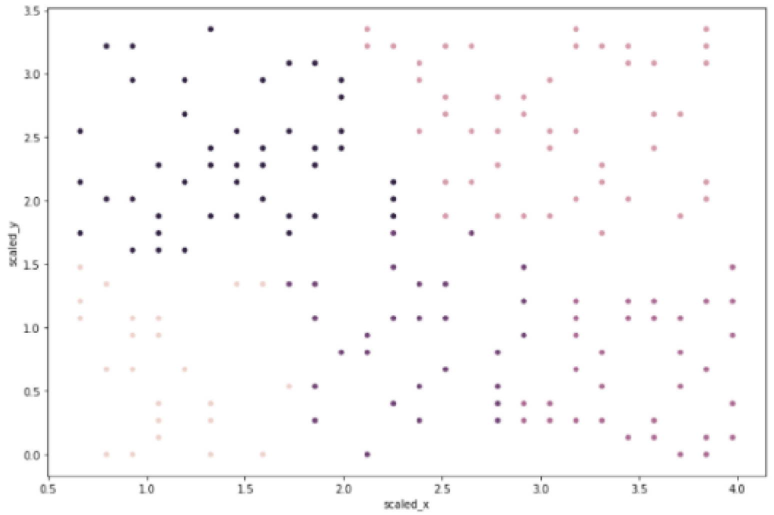
Limitations of k-means clustering

1. The elbow method is one way to determine the right k (number of clusters), but may not always work.
2. The impact of seeds on clustering, as defining the initial cluster centers is a random process.

Seed: `np.array(1000, 2000)`

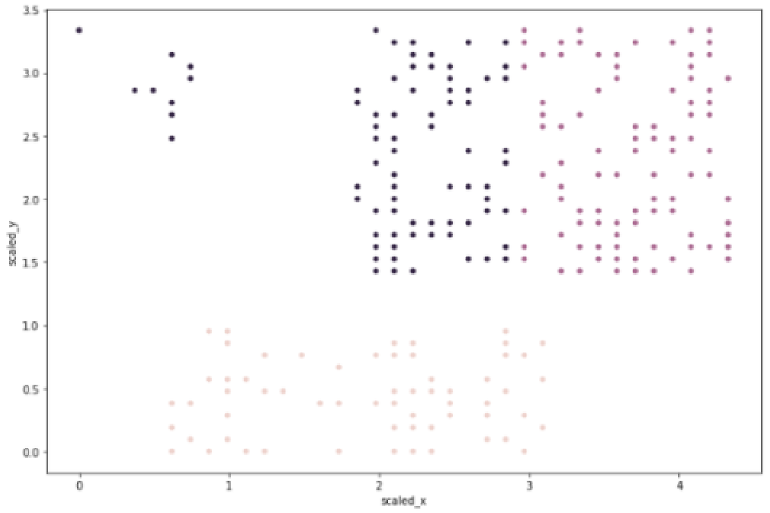


Seed: `np.array(1,2,3)`

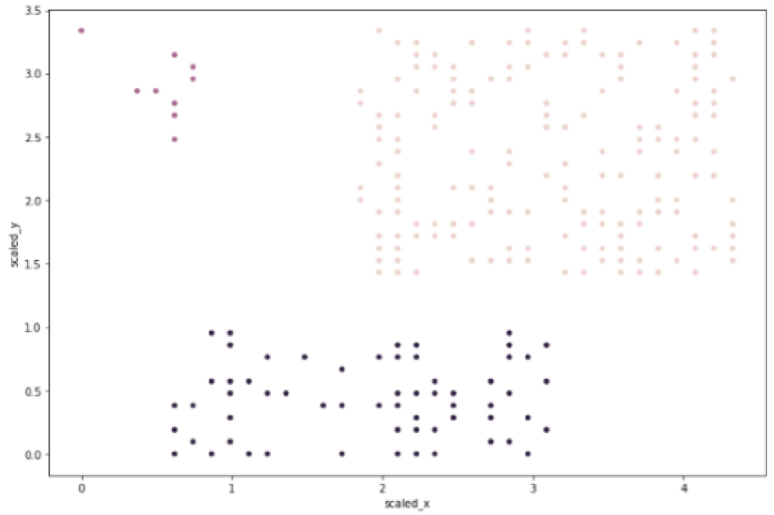


3. The formation of equal-sized clusters, biased towards equal sized clusters.

K-means clustering with 3 clusters



Hierarchical clustering with 3 clusters



Each technique has its pros and cons, should kow the underlying assumptions of each technique before applying them. Ideally, you should spend some time pondering over your data size, its patterns and resources and time available to you before finalising on an algorithm.

Clustering is still the exploratory phase of your analysis, so it is perfectly fine for some trial and error.

Impact of seeds on distinct clusters

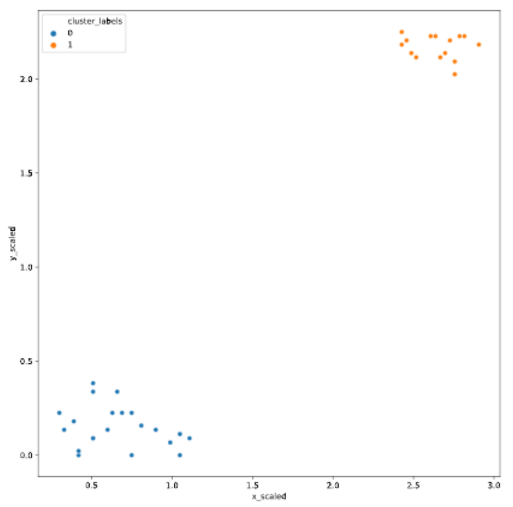
You noticed the impact of seeds on a dataset that did not have well-defined groups of clusters. In this exercise, you will explore whether seeds impact the clusters in the Comic Con data, where the clusters are well-defined.

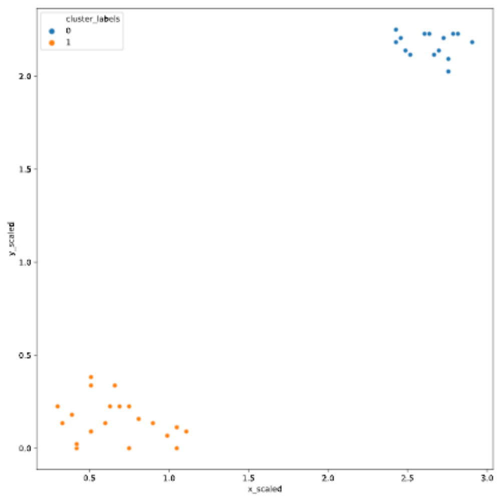
The data is stored in a Pandas data frame, `comic_con`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of people at a given point in time.

```
# Import random class
from numpy import random

# Initialize seed
random.seed(0)

# Run kmeans clustering
cluster_centers, distortion = kmeans(comic_con[['x_scaled', 'y_scaled']], 2)
comic_con['cluster_labels'], distortion_list = vq(comic_con[['x_scaled', 'y_scaled']], cluster_centers)

# Plot the scatterplot
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = comic_con)
plt.show()
```

```
# Initialize seed
random.seed([1, 2, 1000])
```

Notice that the plots have not changed after changing the seed as the clusters are well-defined.



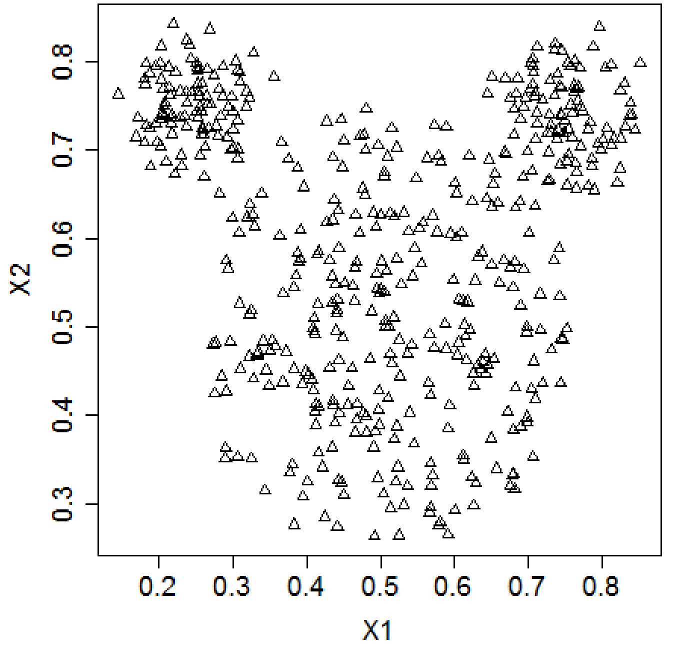


Uniform clustering patterns

Now that you are familiar with the impact of seeds, let us look at the bias in k-means clustering towards the formation of uniform clusters.

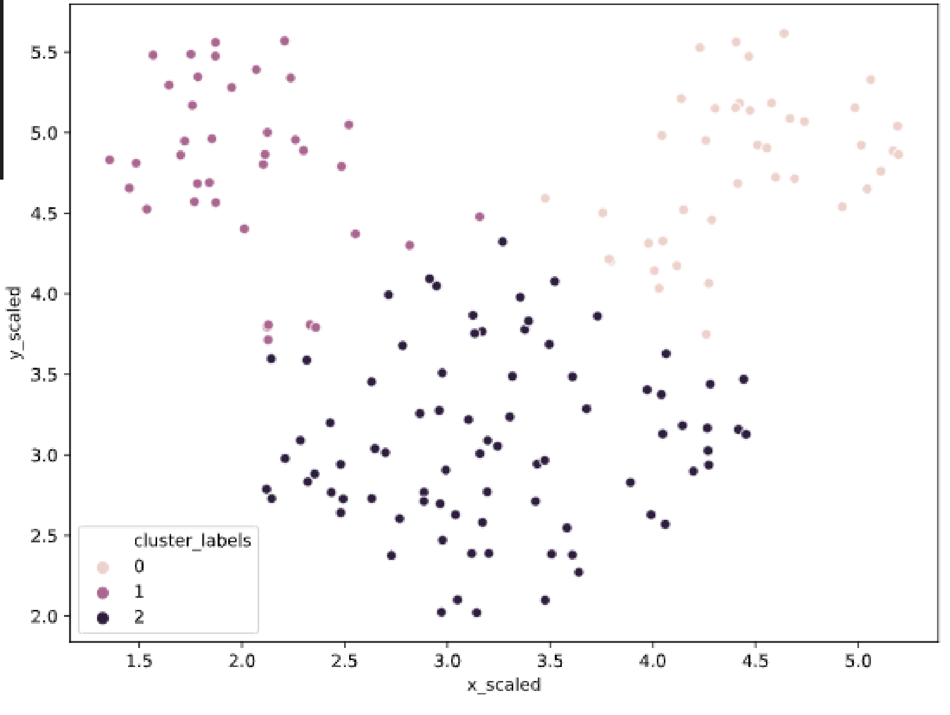
Let us use a mouse-like dataset for our next exercise. A mouse-like dataset is a group of points that resemble the head of a mouse: it has three clusters of points arranged in circles, one each for the face and two ears of a mouse.

Here is how a typical mouse-like dataset looks like (Source).



The data is stored in a Pandas data frame, `mouse`. `x_scaled` and `y_scaled` are the column names of the standardized X and Y coordinates of the data points.

```
# Import the kmeans and vq functions
from scipy.cluster.vq import kmeans, vq

# Generate cluster centers
cluster_centers, distortion = kmeans(mouse[['x_scaled', 'y_scaled']], 3)

# Assign cluster labels
mouse['cluster_labels'], distortion_list = vq(mouse[['x_scaled', 'y_scaled']], cluster_centers)

# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = mouse)
plt.show()
```



Notice that k-means is unable to capture the three visible clusters clearly, and the two clusters towards the top have taken in some points along the boundary.

This happens due to the underlying assumption in kmeans algorithm to minimize distortions which leads to clusters that are similar in terms of area.

FIFA 18: defenders revisited

In the FIFA 18 dataset, various attributes of players are present. Two such attributes are:

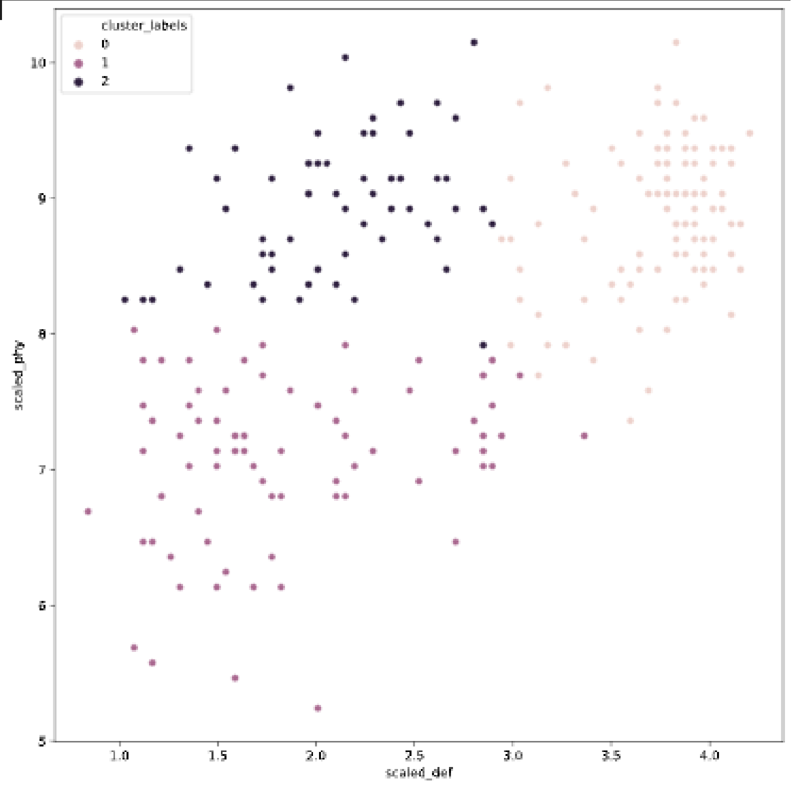
- *defending*: a number which signifies the defending attributes of a player
- *physical*: a number which signifies the physical attributes of a player

These are typically defense-minded players. In this exercise, you will perform clustering based on these attributes in the data.

The following modules have been pre-loaded: `kmeans`, `vq` from `scipy.cluster.vq`, `matplotlib.pyplot` as `plt`, `seaborn` as `sns`. The data for this exercise is stored in a Pandas dataframe, `fifa`. The scaled variables are `scaled_def` and `scaled_phy`.

```python
# Set up a random seed in numpy
random.seed([1000,2000])

# Fit the data into a k-means algorithm
cluster_centers, _ = kmeans(fifa[['scaled_def', 'scaled_phy']], 3)

# Assign cluster labels
fifa['cluster_labels'], _ = vq(fifa[['scaled_def', 'scaled_phy']], cluster_centers)

# Display cluster centers
print(fifa[['scaled_def', 'scaled_phy', 'cluster_labels']].groupby('cluster_labels').mean())

# Create a scatter plot through seaborn
sns.scatterplot(x='scaled_def', y='scaled_phy', hue='cluster_labels', data=fifa)
plt.show()
```

```
<script.py> output:
                       scaled_def  scaled_phy
        cluster_labels
        0                    3.74        8.87
        1                    1.87        7.08
        2                    2.10        8.94
```

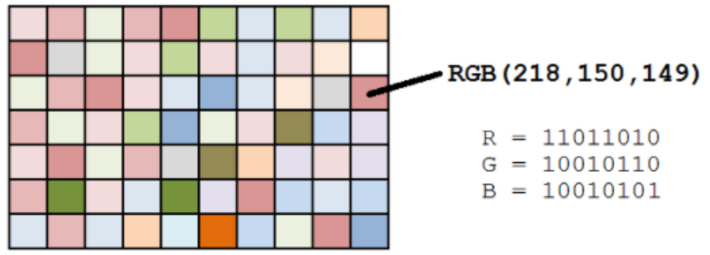


Notice that the seed has an impact on clustering as the data is uniformly distributed.

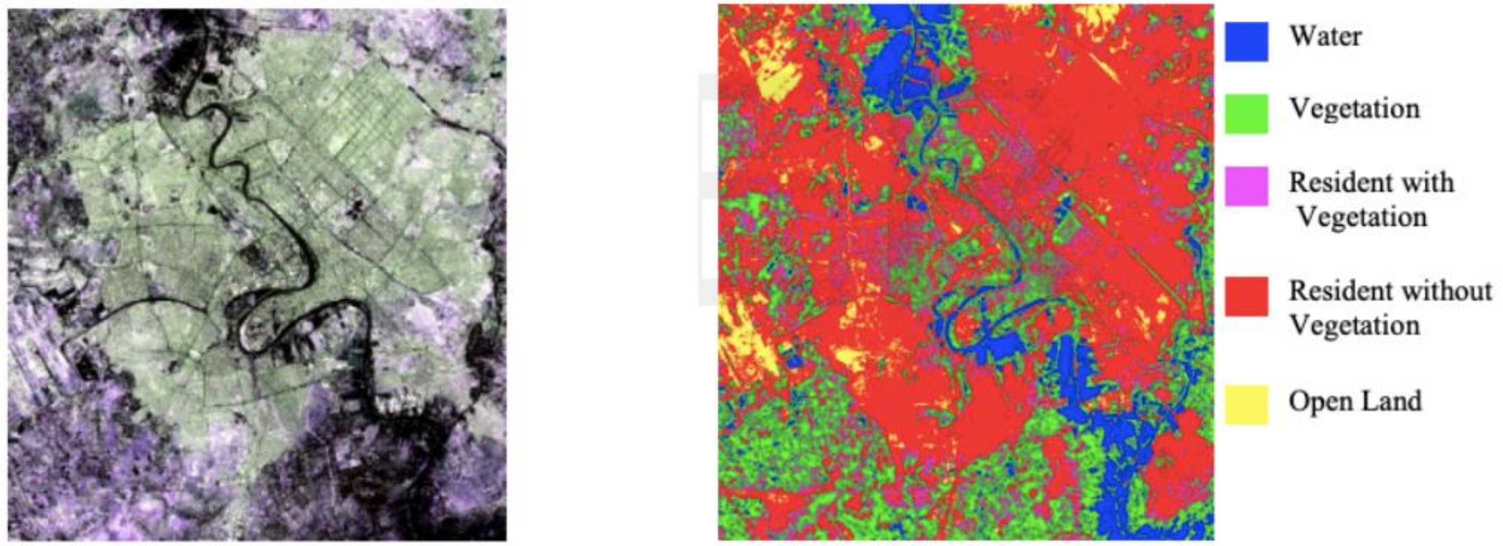
Chapter 4. Clustering in Real World

Now that you are familiar with two of the most popular clustering techniques, this chapter helps you apply this knowledge to real-world problems. The chapter first discusses the process of finding dominant colors in an image, before moving on to the problem discussed in the introduction - clustering of news articles. The chapter concludes with a discussion on clustering with multiple variables, which makes it difficult to visualize all the data.

Dominant colors in images



Perform k-means on standardised RGB values to find cluster centers, eg. identify features in satellite images.



Tools to find dominant colors

- Convert image to pixels: `matplotlib.image.imread`
- Display colors of cluster centers: `matplotlib.pyplot.imshow`

```
import matplotlib.image as img
image = img.imread('sea.jpg')
image.shape
```

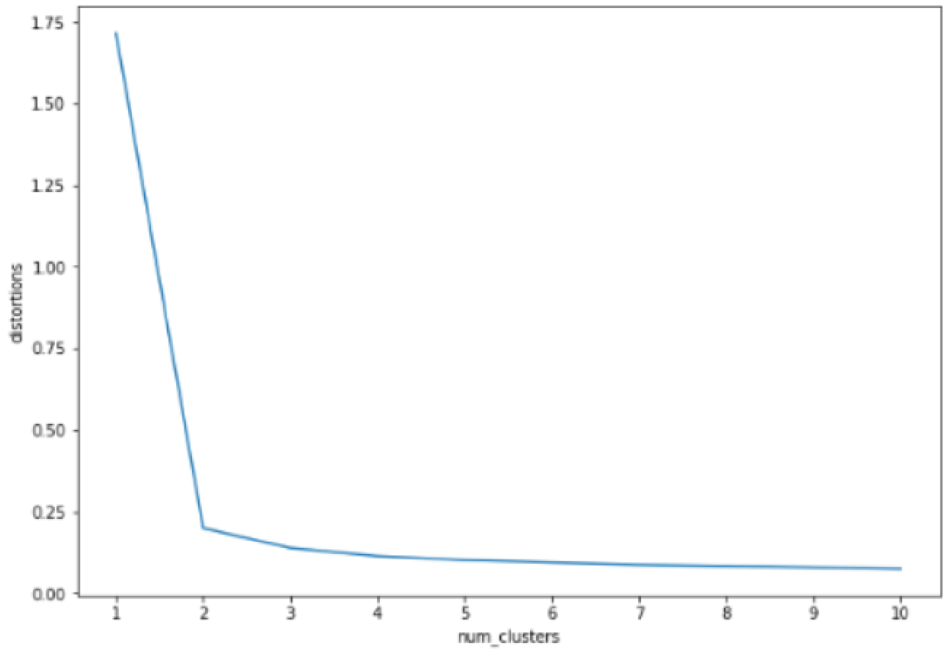
```
(475, 764, 3)
```

```
r = []
g = []
b = []

for row in image:
    for pixel in row:
        # A pixel contains RGB values
        temp_r, temp_g, temp_b = pixel
        r.append(temp_r)
        g.append(temp_g)
        b.append(temp_b)
```

```
pixels = pd.DataFrame({'red': r,
                       'blue': b,
                       'green': g})
pixels.head()
```

red	blue	green
252	255	252
75	103	81
...	...	...

```
distortions = []
num_clusters = range(1, 11)

# Create a list of distortions from the kmeans method
for i in num_clusters:
    cluster_centers, _ = kmeans(pixels[['scaled_red', 'scaled_blue',
                                        'scaled_green']], i)
    distortions.append(distortion)

# Create a data frame with two lists - number of clusters and distortions
elbow_plot = pd.DataFrame({'num_clusters': num_clusters,
                           'distortions': distortions})

# Creat a line plot of num_clusters and distortions
sns.lineplot(x='num_clusters', y='distortions', data = elbow_plot)
plt.xticks(num_clusters)
plt.show()
```



The elbow plot shows that there are 2 dominant colors in the image.

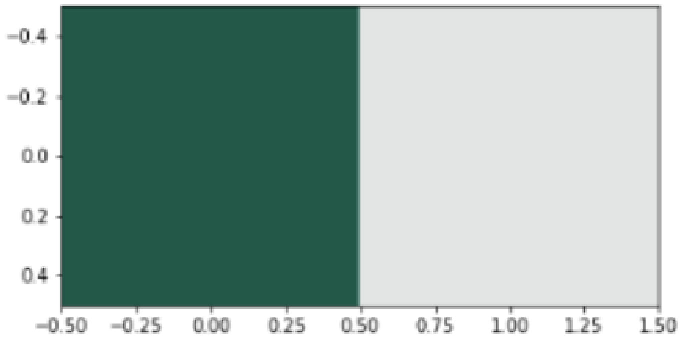
```
cluster_centers, _ = kmeans(pixels[['scaled_red', 'scaled_blue',
                                    'scaled_green']], 2)
```

```
colors = []

# Find Standard Deviations
r_std, g_std, b_std = pixels[['red', 'blue', 'green']].std()

# Scale actual RGB values in range of 0-1
for cluster_center in cluster_centers:
    scaled_r, scaled_g, scaled_b = cluster_center
    colors.append((
        scaled_r * r_std/255,
        scaled_g * g_std/255,
        scaled_b * b_std/255
    ))
```

```
#Dimensions: 2 x 3 (N X 3 matrix)
print(colors)
```

```
[(0.08192923122023911, 0.34205845943857993, 0.2824002984155429),
 (0.893281510956742, 0.899818770315129, 0.8979114272960784)]
```

```
#Dimensions: 1 x 2 x 3 (1 X N x 3 matrix)
plt.imshow([colors])
plt.show()
```



Extract RGB values from image

There are broadly three steps to find the dominant colors in an image:

- Extract RGB values into three lists.
- Perform k-means clustering on scaled RGB values.
- Display the colors of cluster centers.

To extract RGB values, we use the `imread()` function of the `image` class of `matplotlib`. Empty lists, `r`, `g` and `b` have been initialized.

For the purpose of finding dominant colors, we will be using the following image:

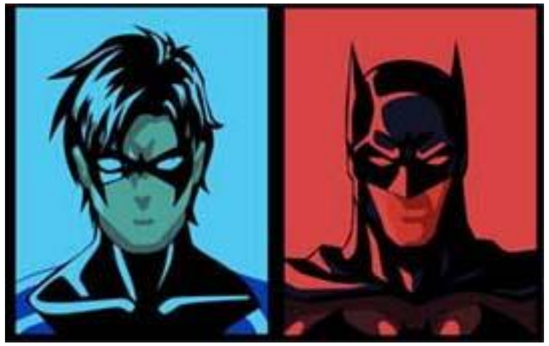
```python
# Import image class of matplotlib
import matplotlib.image as img

# Read batman image and print dimensions
batman_image = img.imread('batman.jpg')
print(batman_image.shape)

# Store RGB values of all pixels in lists r, g and b
for row in batman_image:
    for temp_r, temp_g, temp_b in row:
        r.append(temp_r)
        g.append(temp_g)
        b.append(temp_b)
```

```
<script.py> output:
    (57, 90, 3)
```

You have successfully extracted the RGB values of the image into three lists, one for each color channel.

How many dominant colors?

We have loaded the following image using the `imread()` function of the `image` class of `matplotlib`.

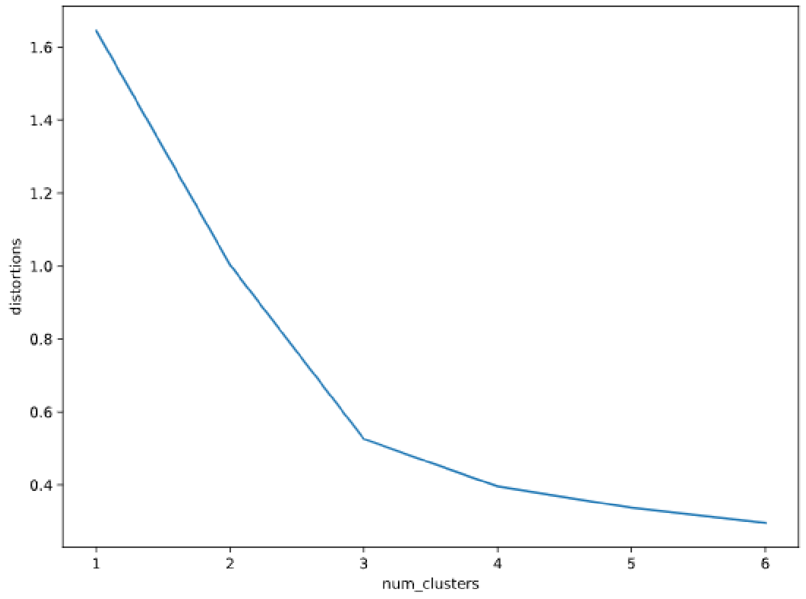


The RGB values are stored in a data frame, `batman_df`. The RGB values have been standardized used the `whiten()` function, stored in columns, `scaled_red`, `scaled_blue` and `scaled_green`.

Construct an elbow plot with the data frame. How many dominant colors are present?

```
distortions = []
num_clusters = range(1, 7)

# Create a list of distortions from the kmeans function
for i in num_clusters:
    cluster_centers, distortion = kmeans(batman_df[['scaled_red', 'scaled_blue', 'scaled_green']], i)
    distortions.append(distortion)

# Create a data frame with two lists, num_clusters and distortions
elbow_plot = pd.DataFrame({'num_clusters': num_clusters, 'distortions': distortions})

# Create a line plot of num_clusters and distortions
sns.lineplot(x='num_clusters', y='distortions', data = elbow_plot)
plt.xticks(num_clusters)
plt.show()
```

Notice that there are three distinct colors present in the image, which is supported by the elbow plot.



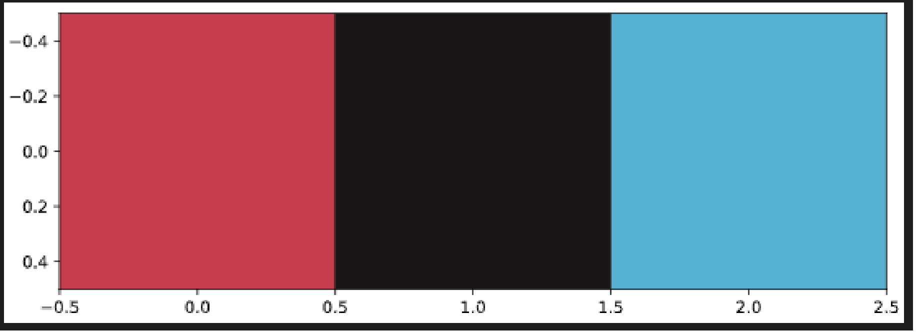
Display dominant colors

We have loaded the following image using the `imread()` function of the `image` class of `matplotlib`.

To display the dominant colors, convert the colors of the cluster centers to their raw values and then converted them to the range of 0-1, using the following formula: `converted_pixel = standardized_pixel * pixel_std / 255`

The RGB values are stored in a data frame, `batman_df`. The scaled RGB values are stored in columns, `scaled_red`, `scaled_blue` and `scaled_green`. The cluster centers are stored in the variable `cluster_centers`, which were generated using the `kmeans()` function with three clusters.

```
# Get standard deviations of each color
r_std, g_std, b_std = batman_df[['red', 'green', 'blue']].std()

for cluster_center in cluster_centers:
    scaled_r, scaled_g, scaled_b = cluster_center
    # Convert each standardized value to scaled value
    colors.append((
        scaled_r * r_std / 255,
        scaled_g * g_std / 255,
        scaled_b * b_std / 255
    ))

# Display colors of cluster centers
plt.imshow([colors])
plt.show()
```



Notice the three colors resemble the three that are indicative from visual inspection of the image.

Document clustering

1. convert text into smaller parts called tokens, using **NLTK's** word_tokenize method.

2. clean data for anything that does not add value to analysis, eg. punctuations, emoticons, and stop words (is, the, are, etc).

3. find the TF-IDF of the terms, or a weighted statistic that describes the importance of a term in a document.

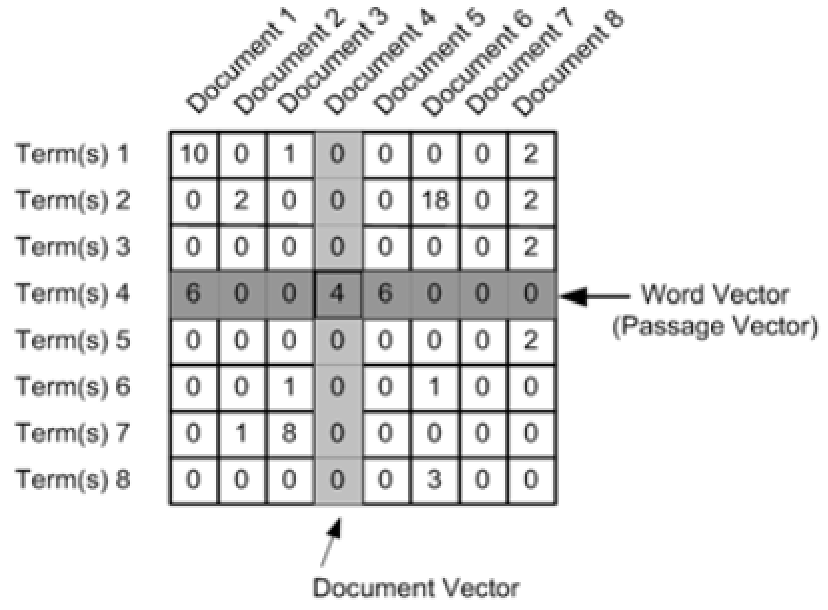
4. cluster the TF-IDF matrix.

5. display the top terms in each cluster.

```
from nltk.tokenize import word_tokenize
import re
```

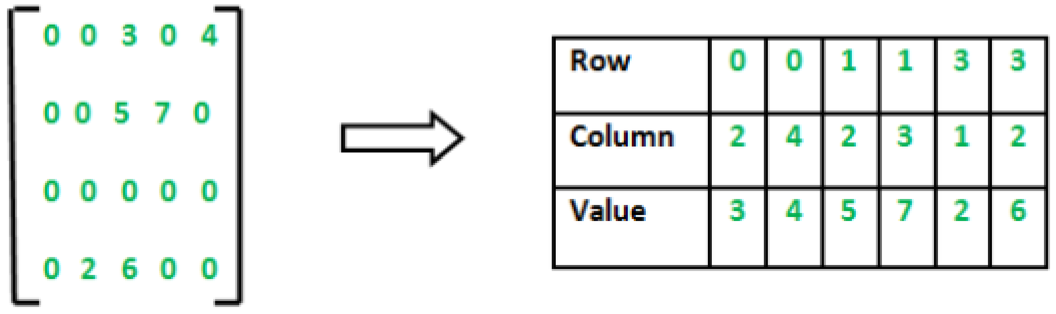
```python
def remove_noise(text, stop_words = []):
    tokens = word_tokenize(text)
    cleaned_tokens = []
    for token in tokens:
        token = re.sub('[^A-Za-z0-9]+', '', token)
        if len(token) > 1 and token.lower() not in stop_words:
            # Get lowercase
            cleaned_tokens.append(token.lower())
    return cleaned_tokens
remove_noise("It is lovely weather we are having.
              I hope the weather continues.")
```

```
['lovely', 'weather', 'hope', 'weather', 'continues']
```



	Document 1	Document 2	Document 3	Document 4	Document 5	Document 6	Document 7	Document 8
Term(s) 1	10	0	1	0	0	0	0	2
Term(s) 2	0	2	0	0	0	18	0	2
Term(s) 3	0	0	0	0	0	0	0	2
Term(s) 4	6	0	0	4	6	0	0	0
Term(s) 5	0	0	0	0	0	0	0	2
Term(s) 6	0	0	1	0	0	1	0	0
Term(s) 7	0	1	8	0	0	0	0	0
Term(s) 8	0	0	0	0	0	3	0	0

Word Vector
(Passage Vector)
Document Vector



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Document-Term matrix

Sparse matrix

TF-IDF (Term Frequency - Inverse Document Frequency)

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=50,
                                   min_df=0.2, tokenizer=remove_noise)
tfidf_matrix = tfidf_vectorizer.fit_transform(data)
```

kmeans() in SciPy does not support sparse matrix, so use .todense() to convert to a matrix.

```
cluster_centers, distortion = kmeans(tfidf_matrix.todense(), num_clusters)
```

```
terms = tfidf_vectorizer.get_feature_names()

for i in range(num_clusters):
    center_terms = dict(zip(terms, list(cluster_centers[i])))
    sorted_terms = sorted(center_terms, key=center_terms.get, reverse=True)
    print(sorted_terms[:3])
```

```
['room', 'hotel', 'staff']

['bad', 'location', 'breakfast']
```

TF-IDF of movie plots

Let us use the plots of randomly selected movies to perform document clustering on. Before performing clustering on documents, they need to be cleaned of any unwanted noise (such as special characters and stop words) and converted into a sparse matrix through TF-IDF of the documents.

Use the `TfidfVectorizer` class to perform the TF-IDF of movie plots stored in the list `plots`. The `remove_noise()` function is available to use as a `tokenizer` in the `TfidfVectorizer` class. The `.fit_transform()` method fits the data into the `TfidfVectorizer` objects and then generates the TF-IDF sparse matrix.

Note: It takes a few seconds to run the `.fit_transform()` method.

```
# Import TfidfVectorizer class from sklearn
from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.75, max_features=50,
                                   min_df=0.1, tokenizer=remove_noise)

# Use the .fit_transform() method on the list plots
tfidf_matrix = tfidf_vectorizer.fit_transform(plots)
```

You have successfully created the sparse matrix. Let us now perform clustering on the matrix.

Top terms in movie clusters

Now that you have created a sparse matrix, generate cluster centers and print the top three terms in each cluster. Use the `.todense()` method to convert the sparse matrix, `tfidf_matrix` to a normal matrix for the `kmeans()` function to process. Then, use the `.get_feature_names()` method to get a list of terms in the `tfidf_vectorizer` object. The `zip()` function in Python joins two lists.

The `tfidf_vectorizer` object and sparse matrix, `tfidf_matrix`, from the previous have been retained in this exercise. `kmeans` has been imported from SciPy.

With a higher number of data points, the clusters formed would be defined more clearly. However, this requires some computational power, making it difficult to accomplish in an exercise here.

```
num_clusters = 2

# Generate cluster centers through the kmeans function
cluster_centers, distortion = kmeans(tfidf_matrix.todense(), num_clusters)

# Generate terms from the tfidf_vectorizer object
terms = tfidf_vectorizer.get_feature_names()

for i in range(num_clusters):
    # Sort the terms and print top 3 terms
    center_terms = dict(zip(terms, list(cluster_centers[i])))
    sorted_terms = sorted(center_terms, key=center_terms.get, reverse=True)
    print(sorted_terms[:3])
```

```
<script.py> output:
    ['father', 'back', 'one']
    ['police', 'man', 'killed']
```

Notice positive, warm words in the first cluster and words referring to action in the second cluster.

Clustering with multiple features

Use plot to visualise 2 features, but not for 3 features and more.

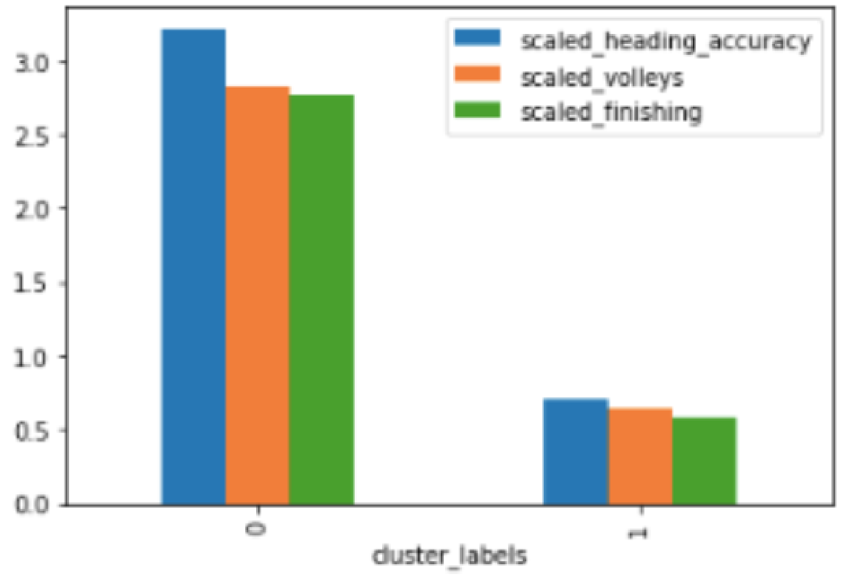
```
# Cluster centers
print(fifa.groupby('cluster_labels')[['scaled_heading_accuracy',
    'scaled_volleys', 'scaled_finishing']].mean())
```

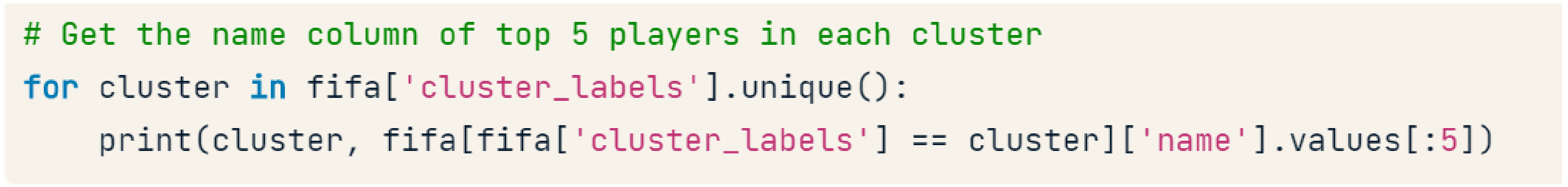
cluster_labels	scaled_heading_accuracy	scaled_volleys	scaled_finishing
0	3.21	2.83	2.76
1	0.71	0.64	0.58

```
# Cluster sizes
print(fifa.groupby('cluster_labels')['ID'].count())
```

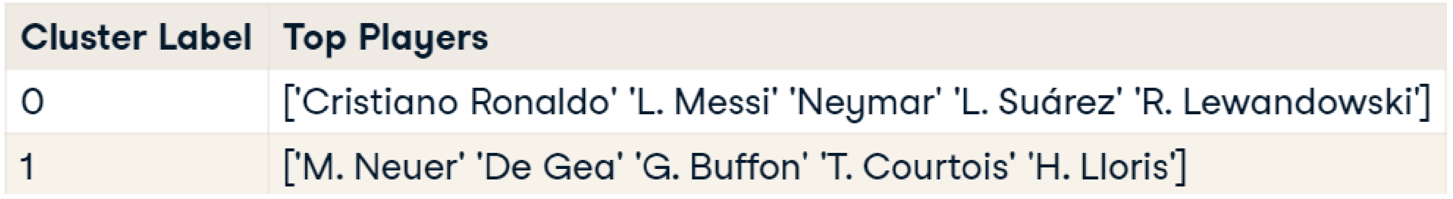
cluster_labels	count
0	886
1	114

```
# Plot cluster centers
fifa.groupby('cluster_labels') \
  [scaled_features].mean()
  .plot(kind='bar')
plt.show()
```





```python
# Get the name column of top 5 players in each cluster
for cluster in fifa['cluster_labels'].unique():
    print(cluster, fifa[fifa['cluster_labels'] == cluster]['name'].values[:5])
```

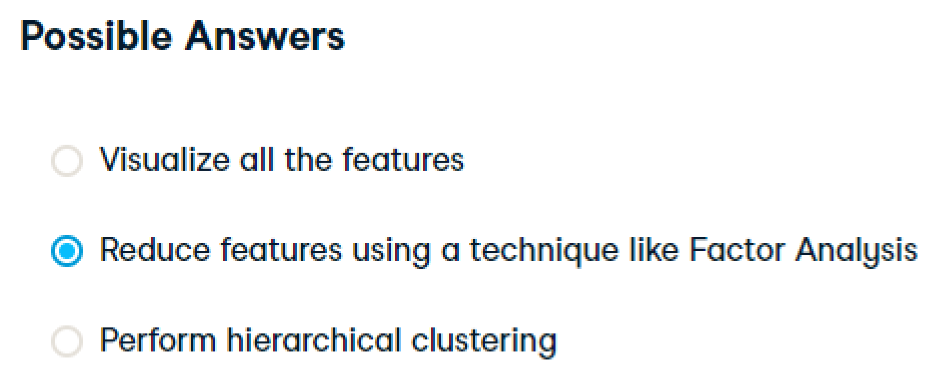


Cluster Label	Top Players
0	['Cristiano Ronaldo' 'L. Messi' 'Neymar' 'L. Suárez' 'R. Lewandowski']
1	['M. Neuer' 'De Gea' 'G. Buffon' 'T. Courtois' 'H. Lloris']

When dealing with a large number of features, certain techniques of feature reduction may be used:
1. Factor analysis
2. Multidimensional scaling

Clustering with many features

What should you do if you have too many features for clustering?



Possible Answers

- Visualize all the features
- [✓ Reduce features using a technique like Factor Analysis]
- Perform hierarchical clustering

Answer: You should explore steps to reduce the number of features.

Basic checks on clusters

In the FIFA 18 dataset, we have concentrated on defenders in previous exercises. Let us try to focus on attacking attributes of a player. Pace (`pac`), Dribbling (`dri`) and Shooting (`sho`) are features that are present in attack minded players. In this exercise, k-means clustering has already been applied on the data using the scaled values of these three attributes. Try some basic checks on the clusters so formed.

The data is stored in a Pandas data frame, `fifa`. The scaled column names are present in a list `scaled_features`. The cluster labels are stored in the `cluster_labels` column. Recall the `.count()` and `.mean()` methods in Pandas help you find the number of observations and mean of observations in a data frame.

```python
# Print the size of the clusters
print(fifa.groupby('cluster_labels')['ID'].count())

# Print the mean value of wages in each cluster
print(fifa.groupby('cluster_labels')['eur_wage'].mean())
```

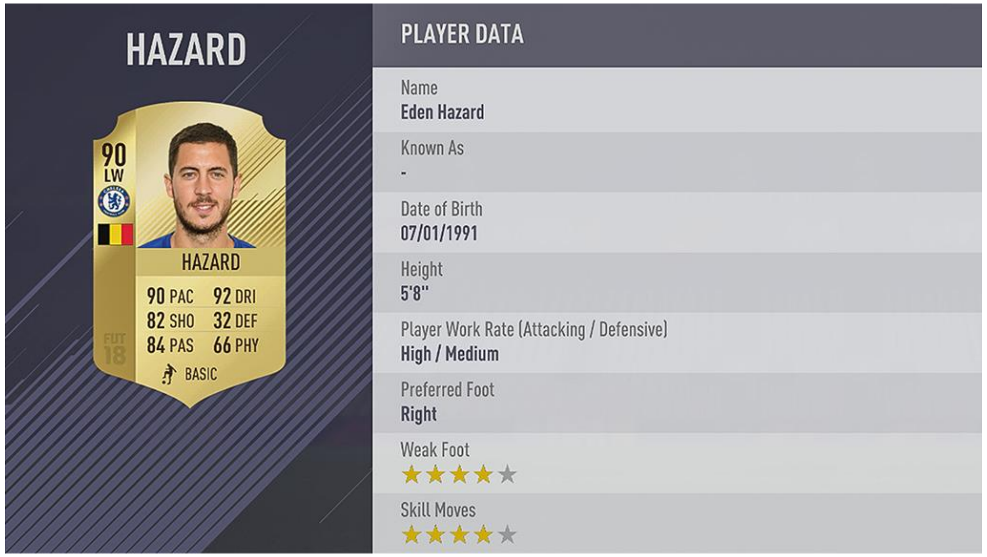
```
<script.py> output:
    cluster_labels
    0      83
    1     107
    2      60
    Name: ID, dtype: int64
    cluster_labels
    0    132108.43
    1    130308.41
    2    117583.33
    Name: eur_wage, dtype: float64
```

In this example, the cluster sizes are not very different, and there are no significant differences that can be seen in the wages. Further analysis is required to validate these clusters.

FIFA 18: what makes a complete player?

The overall level of a player in FIFA 18 is defined by six characteristics:
pace (`pac`),
shooting (`sho`),
passing (`pas`),
dribbling (`dri`),
defending (`def`),
physical (`phy`).

Here is a sample card:



PLAYER DATA

Name
Eden Hazard

Known As
-

Date of Birth
07/01/1991

Height
5'8"

Player Work Rate (Attacking / Defensive)
High / Medium

Preferred Foot
Right

Weak Foot
★★★★★

Skill Moves
★★★★★

In this exercise, you will use all six characteristics to create clusters. The data for this exercise is stored in a Pandas dataframe, `fifa`. `features` is the list of these column names and `scaled_features` is the list of columns which contains their scaled values. The following have been pre-loaded: `kmeans`, `vq` from `scipy.cluster.vq`, `matplotlib.pyplot` as `plt`, `seaborn` as `sns`.

Before you start the exercise, you may wish to explore `scaled_features` in the console to check out the list of six scaled columns names.

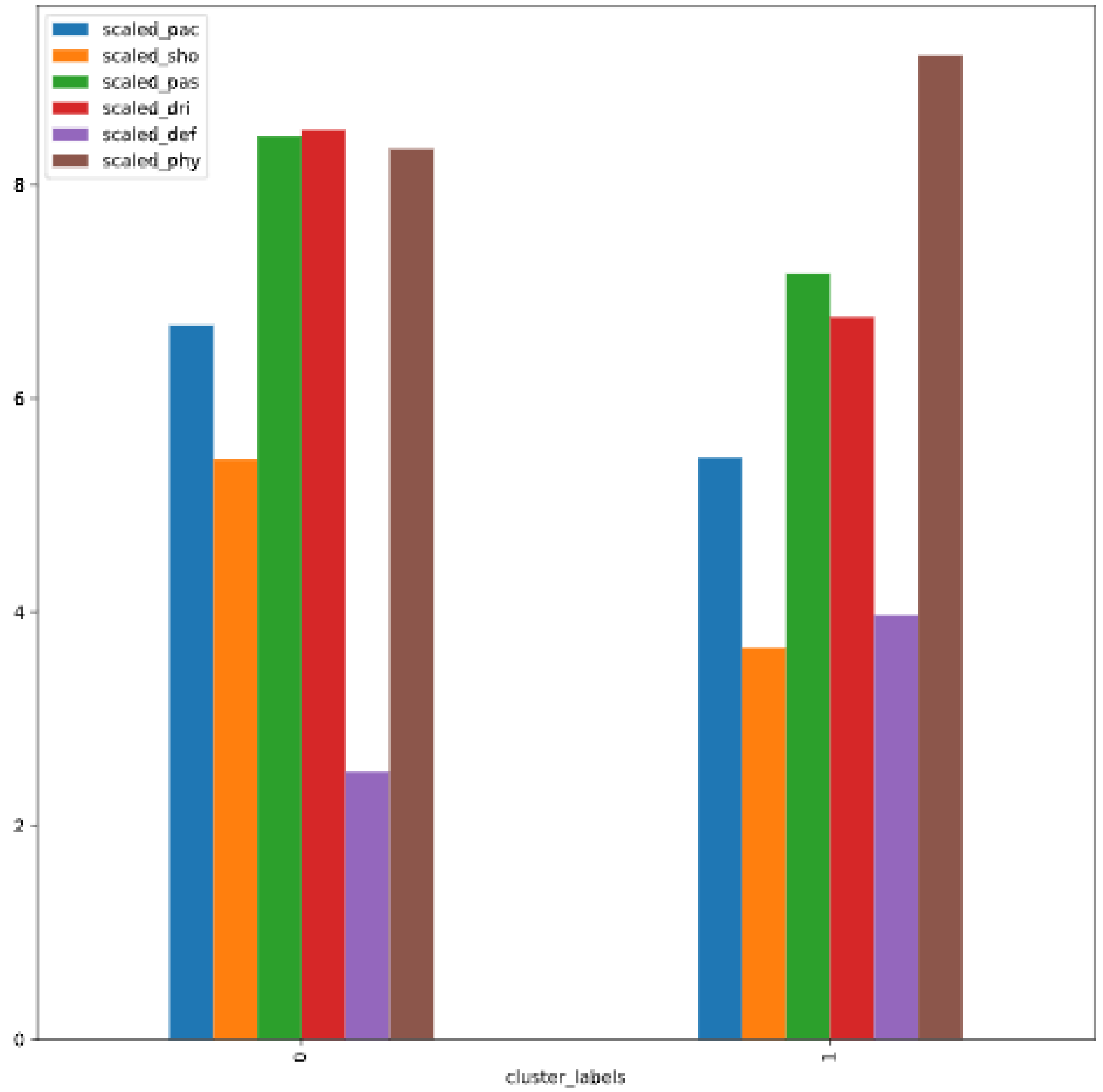
```python
# Create centroids with kmeans for 2 clusters
cluster_centers, _ = kmeans(fifa[scaled_features], 2)

# Assign cluster labels and print cluster centers
fifa['cluster_labels'], _ = vq(fifa[scaled_features], cluster_centers)
print(fifa.groupby('cluster_labels')[scaled_features].mean())

# Plot cluster centers to visualize clusters
fifa.groupby('cluster_labels')[scaled_features].mean().plot(legend=True, kind='bar')
plt.show()

# Get the name column of first 5 players in each cluster
for cluster in fifa['cluster_labels'].unique():
    print(cluster, fifa[fifa['cluster_labels'] == cluster]['name'].values[:5])
```

```
<script.py> output:
                    scaled_pac  scaled_sho  scaled_pas  scaled_dri  scaled_def  \
    cluster_labels
    0                     6.68        5.43        8.46        8.51        2.50
    1                     5.44        3.66        7.17        6.76        3.97

                    scaled_phy
    cluster_labels
    0                     8.34
    1                     9.21
    0 ['Cristiano Ronaldo' 'L. Messi' 'Neymar' 'L. Suárez' 'M. Neuer']
    1 ['Sergio Ramos' 'G. Chiellini' 'D. Godín' 'Thiago Silva' 'M. Hummels']
```

The data was sorted before you performed the clustering.

Notice the top players in each cluster are representative of the overall characteristics of the cluster - one of the clusters primarily represents attackers, whereas the other represents defenders.

Surprisingly, a top goalkeeper Manuel Neuer is seen in the attackers group, but he is known for going out of the box and participating in open play, which are reflected in his FIFA 18 attributes.



Course completed!

Recap topics covered:

- Clustering as one of the exploratory steps
- Basics of unsupervised learning
- Data preparation for cluster analysis
- Normalization of data
- Hierarchical clustering: ward method, single method, complete method
- Visualisation of clusters: matplotlib, seaborn
- Using SciPy dendrogram to determine how many clusters
- Limitations of hierarchical clustering - runtime
- K-means clustering
- Using Elbow Plot to determine how many clusters
- Limitations of k-means clustering
- Uniform clustering patterns
- Extracting RGB values and clustering dominant colors in images
- Using nltk's word_tokenize to cluster document
- Clustering with 3 or more features

Happy learning!