Untitled

June 7, 2020

1 Cluster Analysis in Python

Cluster analysis is a branch of unsupervised learning where data is not labeled and machine turns the data into different categories based on patterns found in the data. Some of the examples of cluster analysis and unsupervised learning can be classification of news articles by Google and segmentation of customers based on their spending habits etc.

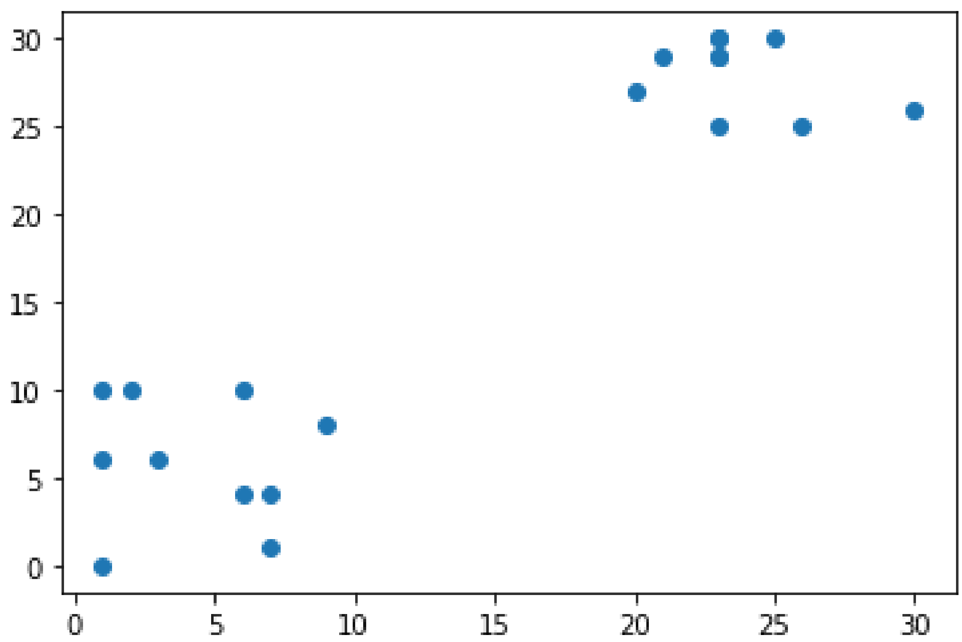
[171]:
```
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.cluster.hierarchy import linkage, fcluster
from scipy.cluster.vq import kmeans, vq
from scipy.cluster.vq import whiten
```

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.

[40]:
```
x = [9, 6, 2, 3, 1, 7, 1, 6, 1, 7, 23, 26, 25, 23, 21, 23, 23, 20, 30, 23]
y = [8, 4, 10, 6, 0, 4, 10, 10, 6, 1, 29, 25, 30, 29, 29, 30, 25, 27, 26, 30]
```

[41]:
```
# Importing plotting class from matplotlib library
from matplotlib import pyplot as plt

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

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



[48]:
```
fb = '/Users/MuhammadBilal/Desktop/Data Camp/Cluster Analysis in Python/Data/df.
↪csv'
```

[53]:
```
df = pd.read_csv(fb)
```

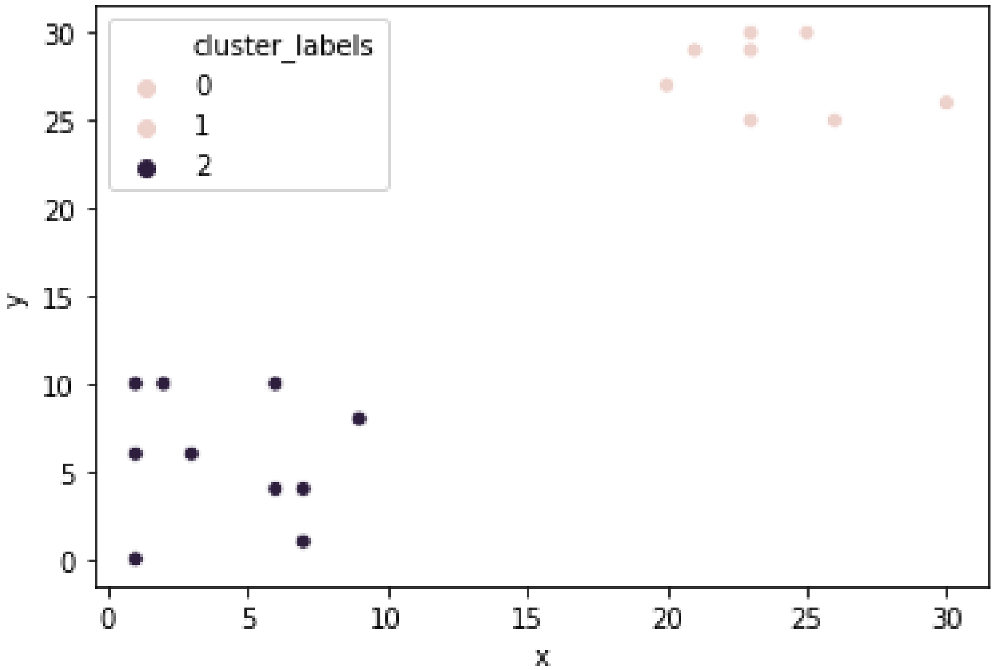
Pokémon sightings: hierarchical clustering

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.

[50]:
```
# Importing linkage and fcluster functions
from scipy.cluster.hierarchy import linkage, fcluster

# Using the linkage() function to compute distance
Z = linkage(df, 'ward')

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

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

```
plt.show()
```



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

Pokémon sightings: k-means clustering

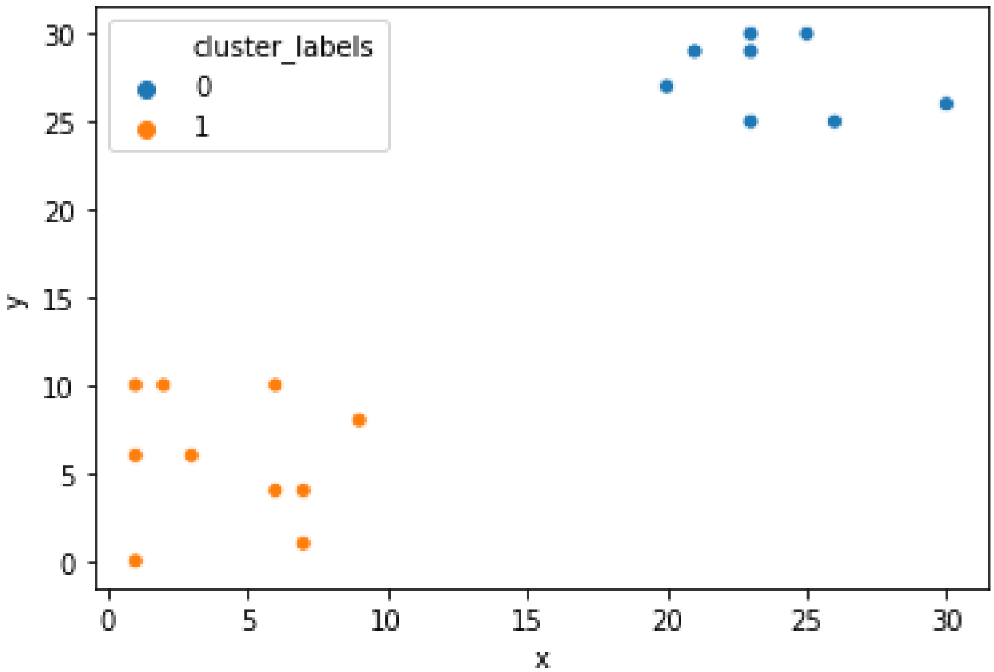
We are going to continue the investigation into the sightings of legendary Pokémon. Just like above, I will use the same example of Pokémon sightings. Here I will form clusters of the sightings using k-means clustering.

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.

[61]:
```
# Converting int dtype to float for kmeans clustering as it takes only float␣
↪dtype.
df = df.astype(float)
```

[62]:
```
# Importing kmeans and vq functions
from scipy.cluster.vq import kmeans, vq

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

# Assigning cluster labels
```

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

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



The results of both types of clustering are similar. I will look at distinctly different results later.

Data preprocessing for clustering analysis

Data preparation for clustering analysis

If we have a set of variables with incomparable units such as the dimensions of a product and its price. Even if variables have the same unit, they may be significantly different in terms of their scales and variances. For example, the amount that one may spend on an inexpensive item like cereals is low as compared to travelling expenses. If we use data in this form the results of clustering may be biased. The clusters formed may be dependent on one variable significantly more than the other. In this case we use normalization. We rescale the values of a variable with respect to standard deviation of the data.

Normalize basic list data

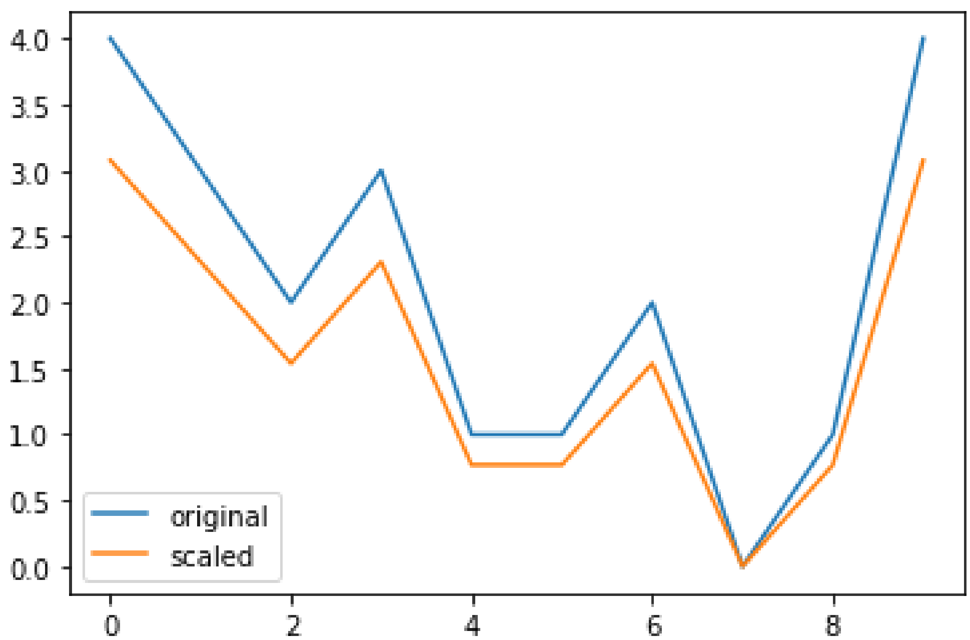
Now that we are aware of normalization, let us try to normalize some data. goals_for is a list of goals scored by a football team in their last ten matches. Let us standardize the data using the whiten() function.

[63]:
```python
# Importing the whiten function
from scipy.cluster.vq import whiten

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

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

```
[3.07692308 2.30769231 1.53846154 2.30769231 0.76923077 0.76923077
 1.53846154 0.         0.76923077 3.07692308]
```

The scaled values have less variations in them. Next I will visualize the data.

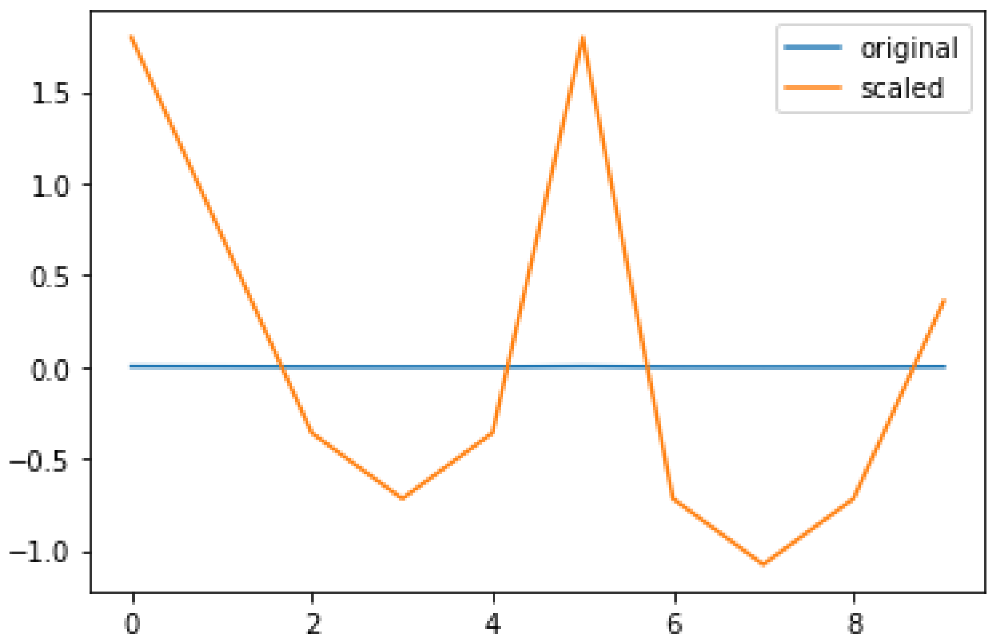
[64]:
```python
# Ploting original data
plt.plot(goals_for, label='original')

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

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

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



The scaled values have lower variations in them.

```
[ ]: Normalization of small numbers

In earlier examples I have normalization of whole numbers. Now I will look at␣
↪the treatment of fractional numbers - the change of interest rates in the␣
↪country of Bangalla over the years.
```

```
[65]: # Preparing data
rate_cuts = [0.0025, 0.001, -0.0005, -0.001, -0.0005, 0.0025, -0.001, -0.0015,␣
↪-0.001, 0.0005]

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

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

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

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



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

I am about to work on contains data on the 1000 top individual players in the game. I will explore various features of the data as I move ahead. I will work with two columns, eur_wage, the wage of a player in Euros and eur_value, their current transfer market value.

The data is stored in a Pandas dataframe, fifa.

[68]:
```
fifa = pd.read_csv('fifa_18_sample_data.csv')
```

[69]:
```
fifa.head()
```

[69]:
```
       ID               name                     full_name  \
0   20801  Cristiano Ronaldo  C. Ronaldo dos Santos Aveiro
1  158023           L. Messi                  Lionel Messi
2  190871             Neymar    Neymar da Silva Santos Jr.
3  176580          L. Suárez                   Luis Suárez
4  167495           M. Neuer                  Manuel Neuer

                  club                                   club_logo  special  age  \
0       Real Madrid CF  https://cdn.sofifa.org/18/teams/243.png     2228   32
1         FC Barcelona  https://cdn.sofifa.org/18/teams/241.png     2158   30
2  Paris Saint-Germain   https://cdn.sofifa.org/18/teams/73.png     2100   25
3         FC Barcelona  https://cdn.sofifa.org/18/teams/241.png     2291   30
4     FC Bayern Munich   https://cdn.sofifa.org/18/teams/21.png     1493   31

                      league  birth_date  height_cm  …  prefers_cb  \
0  Spanish Primera División  1985-02-05      185.0  …       False
1  Spanish Primera División  1987-06-24      170.0  …       False
2            French Ligue 1  1992-02-05      175.0  …       False
3  Spanish Primera División  1987-01-24      182.0  …       False
4         German Bundesliga  1986-03-27      193.0  …       False

  prefers_lb  prefers_lwb prefers_ls prefers_lf prefers_lam  prefers_lcm  \
0      False        False      False      False       False        False
1      False        False      False      False       False        False
2      False        False      False      False       False        False
3      False        False      False      False       False        False
4      False        False      False      False       False        False

   prefers_ldm  prefers_lcb  prefers_gk
0        False        False       False
1        False        False       False
2        False        False       False
3        False        False       False
4        False        False        True

[5 rows x 185 columns]
```

[70]:
```
print(fifa.head())
```

```
       ID               name                    full_name  \
0   20801  Cristiano Ronaldo  C. Ronaldo dos Santos Aveiro
1  158023           L. Messi                 Lionel Messi
2  190871             Neymar    Neymar da Silva Santos Jr.
3  176580          L. Suárez                  Luis Suárez
4  167495           M. Neuer                 Manuel Neuer

                  club                                club_logo  special  age  \
0       Real Madrid CF  https://cdn.sofifa.org/18/teams/243.png     2228   32
1         FC Barcelona  https://cdn.sofifa.org/18/teams/241.png     2158   30
2  Paris Saint-Germain   https://cdn.sofifa.org/18/teams/73.png     2100   25
3         FC Barcelona  https://cdn.sofifa.org/18/teams/241.png     2291   30
4     FC Bayern Munich   https://cdn.sofifa.org/18/teams/21.png     1493   31

                   league  birth_date  height_cm  …  prefers_cb  \
0  Spanish Primera División  1985-02-05      185.0  …       False
1  Spanish Primera División  1987-06-24      170.0  …       False
2            French Ligue 1  1992-02-05      175.0  …       False
3  Spanish Primera División  1987-01-24      182.0  …       False
4          German Bundesliga  1986-03-27      193.0  …       False

  prefers_lb  prefers_lwb prefers_ls prefers_lf prefers_lam  prefers_lcm  \
0      False        False      False      False       False        False
1      False        False      False      False       False        False
2      False        False      False      False       False        False
3      False        False      False      False       False        False
4      False        False      False      False       False        False

   prefers_ldm  prefers_lcb  prefers_gk
0        False        False       False
1        False        False       False
2        False        False       False
3        False        False       False
4        False        False        True

[5 rows x 185 columns]
```

[106]:
```python
for col in fifa.columns:
    print(col)
```

```
ID
name
full_name
club
club_logo
special
age
league
```

```
birth_date
height_cm
weight_kg
body_type
real_face
flag
nationality
photo
eur_value
eur_wage
eur_release_clause
overall
potential
pac
sho
pas
dri
def
phy
international_reputation
skill_moves
weak_foot
work_rate_att
work_rate_def
preferred_foot
crossing
finishing
heading_accuracy
short_passing
volleys
dribbling
curve
free_kick_accuracy
long_passing
ball_control
acceleration
sprint_speed
agility
reactions
balance
shot_power
jumping
stamina
strength
long_shots
aggression
interceptions
positioning
```

vision
penalties
composure
marking
standing_tackle
sliding_tackle
gk_diving
gk_handling
gk_kicking
gk_positioning
gk_reflexes
rs
rw
rf
ram
rcm
rm
rdm
rcb
rb
rwb
st
lw
cf
cam
cm
lm
cdm
cb
lb
lwb
ls
lf
lam
lcm
ldm
lcb
gk
1_on_1_rush_trait
acrobatic_clearance_trait
argues_with_officials_trait
avoids_using_weaker_foot_trait
backs_into_player_trait
bicycle_kicks_trait
cautious_with_crosses_trait
chip_shot_trait
chipped_penalty_trait
comes_for_crosses_trait

corner_specialist_trait
diver_trait
dives_into_tackles_trait
diving_header_trait
driven_pass_trait
early_crosser_trait
fan's_favourite_trait
fancy_flicks_trait
finesse_shot_trait
flair_trait
flair_passes_trait
gk_flat_kick_trait
gk_long_throw_trait
gk_up_for_corners_trait
giant_throw_in_trait
inflexible_trait
injury_free_trait
injury_prone_trait
leadership_trait
long_passer_trait
long_shot_taker_trait
long_throw_in_trait
one_club_player_trait
outside_foot_shot_trait
playmaker_trait
power_free_kick_trait
power_header_trait
puncher_trait
rushes_out_of_goal_trait
saves_with_feet_trait
second_wind_trait
selfish_trait
skilled_dribbling_trait
stutter_penalty_trait
swerve_pass_trait
takes_finesse_free_kicks_trait
target_forward_trait
team_player_trait
technical_dribbler_trait
tries_to_beat_defensive_line_trait
poacher_speciality
speedster_speciality
aerial_threat_speciality
dribbler_speciality
playmaker_speciality
engine_speciality
distance_shooter_speciality
crosser_speciality

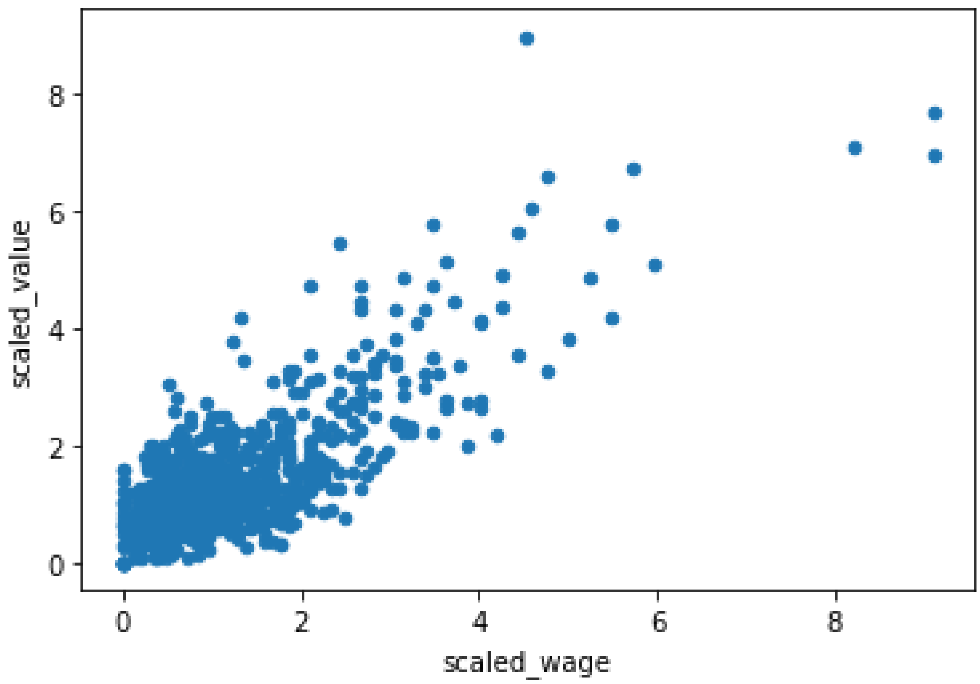
```
free_kick_specialist_speciality
tackling_speciality
tactician_speciality
acrobat_speciality
strength_speciality
clinical_finisher_speciality
prefers_rs
prefers_rw
prefers_rf
prefers_ram
prefers_rcm
prefers_rm
prefers_rdm
prefers_rcb
prefers_rb
prefers_rwb
prefers_st
prefers_lw
prefers_cf
prefers_cam
prefers_cm
prefers_lm
prefers_cdm
prefers_cb
prefers_lb
prefers_lwb
prefers_ls
prefers_lf
prefers_lam
prefers_lcm
prefers_ldm
prefers_lcb
prefers_gk
scaled_wage
scaled_value
scaled_def
scaled_phy
scaled_pac
scaled_dri
scaled_sho
scaled_pas
cluster_labels
```

Preprocessing the data for cluster analysis.

Scaling the values of eur_wage and eur_value using the whiten() function.

[71]:

```
# Scaling wage and value
fifa['scaled_wage'] = whiten(fifa['eur_wage'])
fifa['scaled_value'] = whiten(fifa['eur_value'])
fifa['scaled_def'] = whiten(fifa['def'])
fifa['scaled_phy'] = whiten(fifa['phy'])
fifa['scaled_pac'] = whiten(fifa['pac'])
fifa['scaled_dri'] = whiten(fifa['dri'])
fifa['scaled_sho'] = whiten(fifa['sho'])
fifa['scaled_pas'] = whiten(fifa['pas'])

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

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




```
       scaled_wage  scaled_value
count  1000.000000   1000.000000
mean      1.119812      1.306272
std       1.000500      1.000500
min       0.000000      0.000000
25%       0.467717      0.730412
50%       0.854794      1.022576
75%       1.407184      1.542995
```

```
max        9.112425       8.984064
```

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

Basics of hierarchical clustering

A critical step is to compute the distance matrix at each stage. This is achieved through linkage method. It computes the distance between clusters.

The ward method computes cluster proximity using the difference between summed squares of their joint clusters minus the individual summed squares. The ward method focuses on clusters more concentric towards its center. Once the distance matrix is created, we can create cluster labels through the fcluster method which takes 3 arguments: the distance matrix, the number of clusters and the criteria to form the clusters based on certain threshold.

The single method uses the two closest objects between clusters to determine the inter-cluster proximity. The clusters formed through this method are more dispersed.

The clusters formed by the complete method use the two farthest objects among clusters to determine inter-cluster proximity.

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. Here the data is of last year's footfall, the number of people at the convention ground at a given time. I will decide the location of stall to maximize sales. Using the ward method, applying 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.

[74]:
```python
fp2 = '/Users/MuhammadBilal/Desktop/Data Camp/Cluster Analysis in Python/Data/
 ↪comic_con.csv'
```

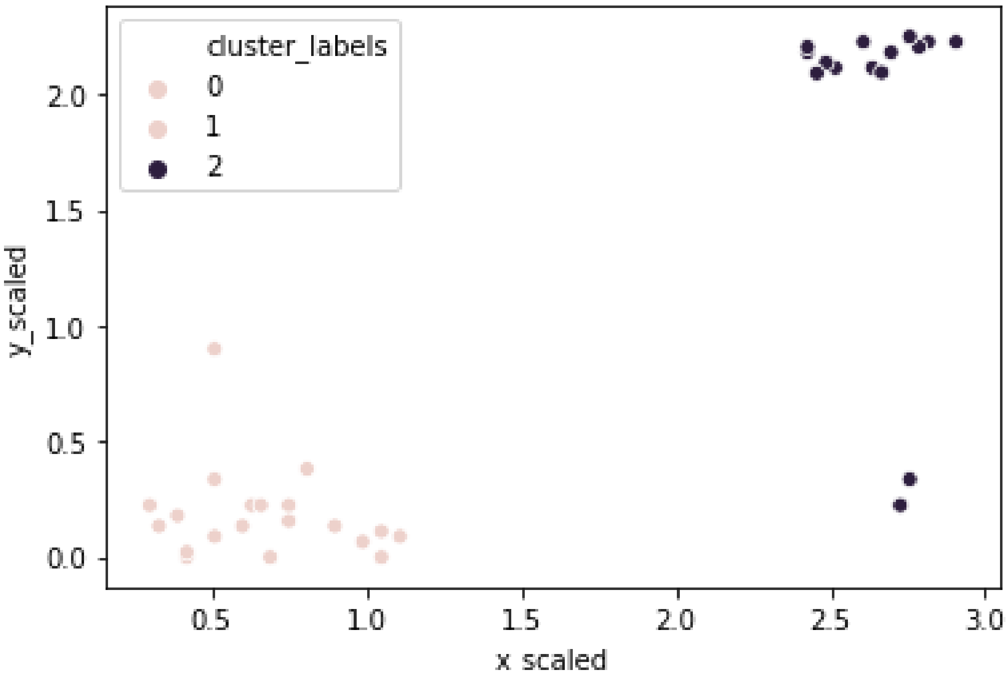
[94]:
```python
comic_con = pd.read_csv(fp2)
comic_con.head()
```

[94]:

	x_coordinate	y_corrdinate	x_scaled	y_scaled
0	17	4	0.509349	0.09001
1	20	6	0.599000	0.13500
2	35	0	1.048000	0.00000
3	14	0	0.419000	0.00000
4	37	4	1.108000	0.09000

[76]:
```python
# Importing the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster, linkage

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

# Assigning cluster labels
```

```
comic_con['cluster_labels'] = fcluster(distance_matrix, 2, criterion='maxclust')

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



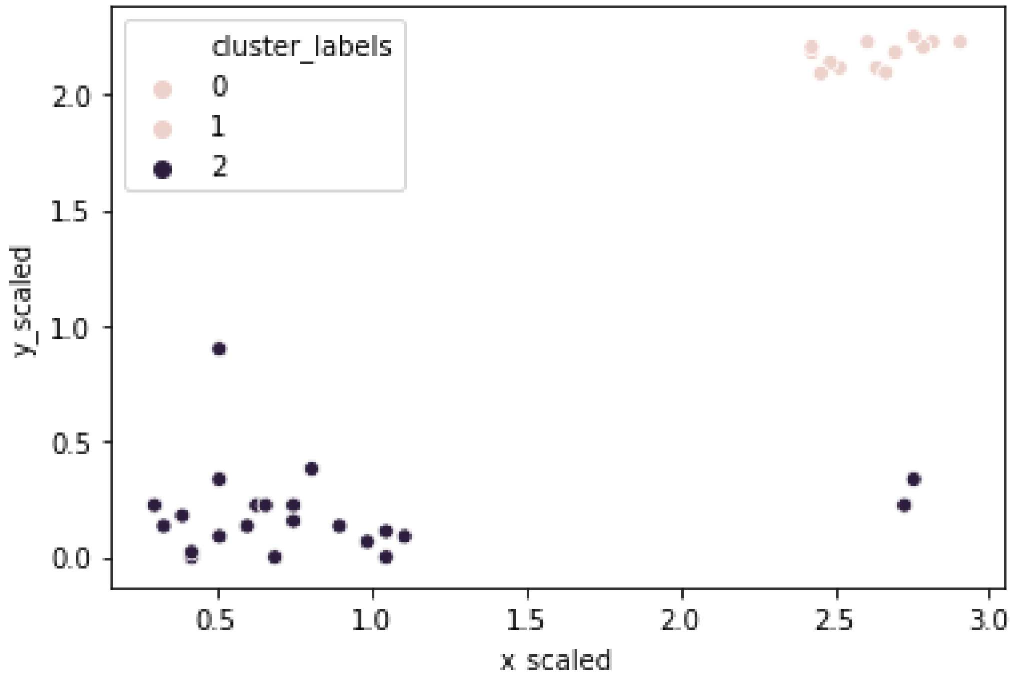
The two clusters correspond to the points of attractions in the figure towards the bottom (a stage) and the top right (an interesting stall).

Hierarchical clustering: single method

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

[77]:
```
# Importing the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster, linkage

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

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

# Plotting clusters
```

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

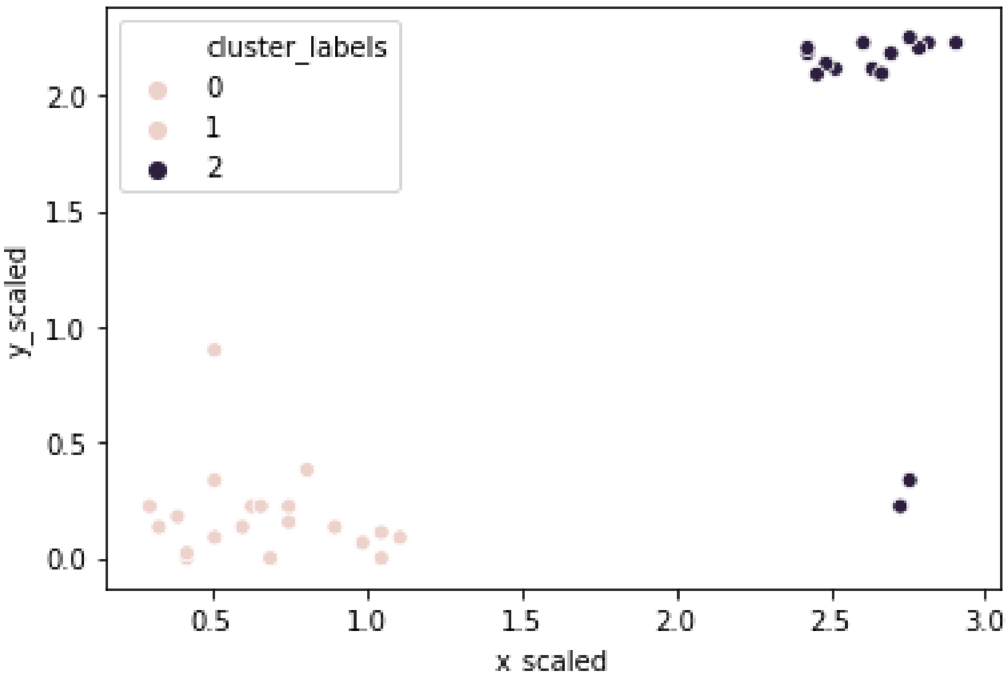


Notice that in this example, the clusters formed are not different from the ones created using the ward method.

Hierarchical clustering: complete method

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

[78]:
```
# Importing the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster, linkage

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

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

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



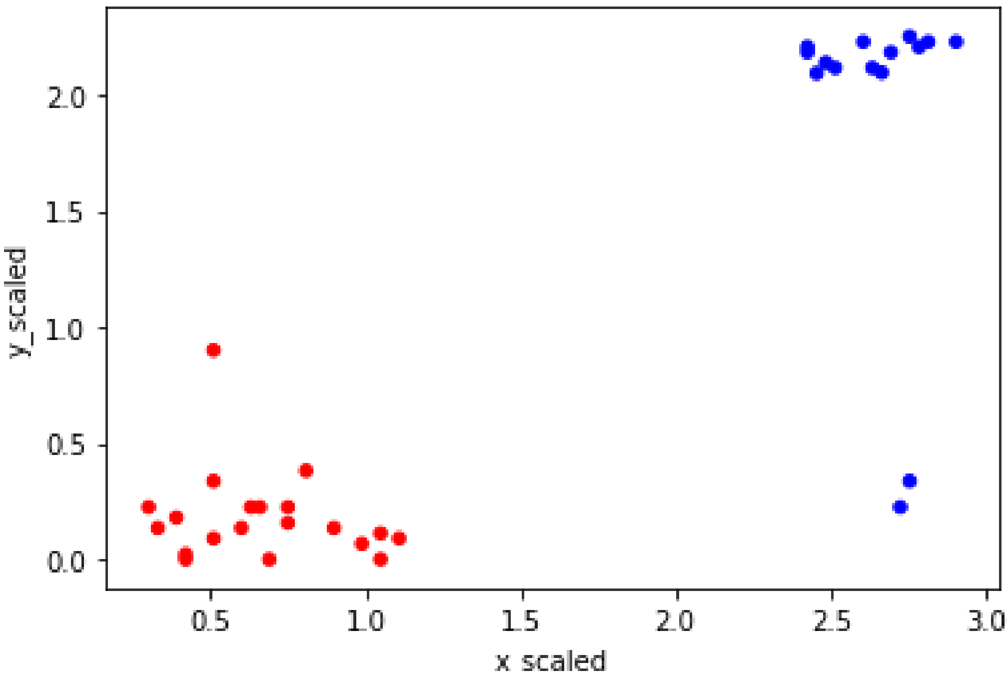
Coincidentally, the clusters formed are not different from the ward or single methods.

Visualizing clusters with matplotlib

Visualizations are necessary to assess the clusters that are formed and spot trends in the data. I will focus on visualizing the footfall dataset from Comic-Con using the matplotlib module.

Cluster_labels has the cluster labels. A linkage object is stored in the variable distance_matrix.

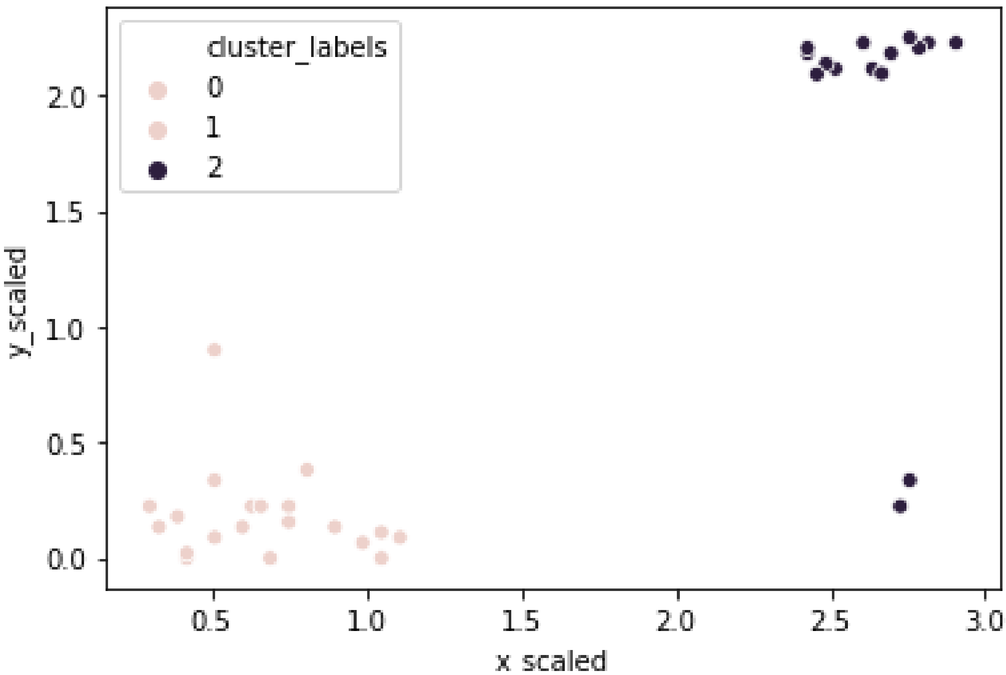
[79]:
```
# Importing the pyplot class
from matplotlib import pyplot as plt

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

# Plotting 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.

[ ]:
```
Visualizing clusters with seaborn

I will 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.
```

[80]:
```
# 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()
```



The legend is automatically shown when using the hue argument.

Visualizing clusters

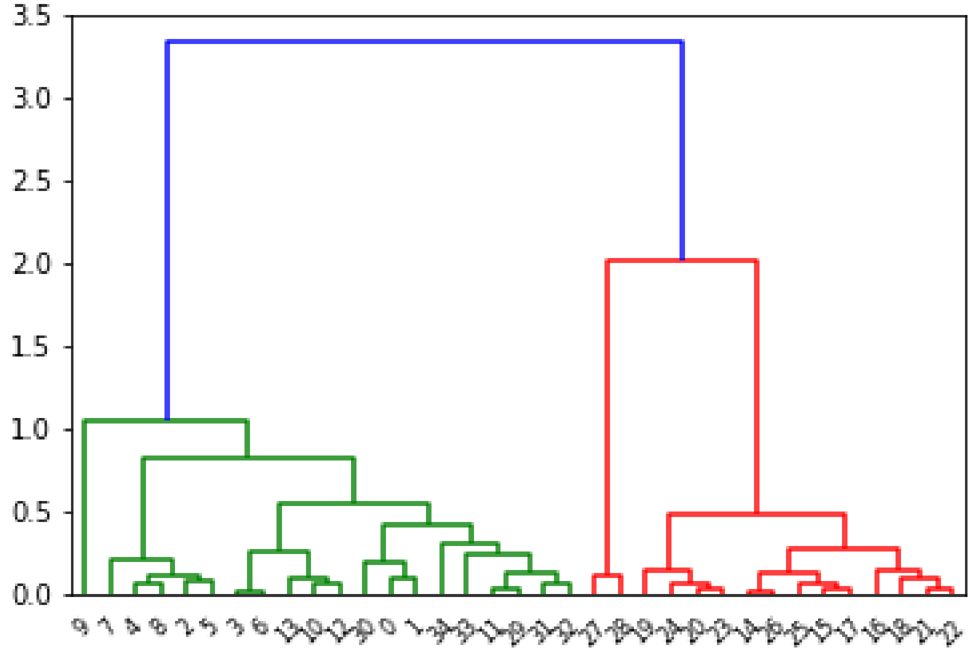
Visualizing clusters can quickly make sense of the clusters formed by any algorithm by visualizing it rather than just looking at cluster centers. It can serve as an additional step for validation of clusters formed. We can also spot trends in the data by visually going through it.

Seaborn provides an argument in its scatterplot method to allow us to use different colors for cluster labels to differentiate the clusters when visualizing them. To decide on the number of clusters in hierarchical clustering we can use a graphical diagram called the dendrogram. A dendrogram is a branching diagram that demonstrates how each cluster is composed by branching out into its child nodes.

Creating a dendrogram

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

[81]:
```
# Importing the dendrogram function
from scipy.cluster.hierarchy import dendrogram

# Creating a dendrogram
dn = dendrogram(distance_matrix)

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



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

Measuring computation time for hierarchical clustering

One of the disadvantages of hierarchical clustering is that it is computationally expensive and it takes too long to run on big datasets.

[82]:
```
%timeit linkage(comic_con[['x_scaled' , 'y_scaled']], method = 'ward' , metric␣
↪= 'euclidean')
```

```
3.26 ms ± 1.61 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[ ]:
```
###### 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. Here I 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 data is stored in a Pandas dataframe, Fifa.
```

[6]:
```
Fifa = pd.read_csv('fifa_18_dataset.csv')
Fifa.head()
```

[6]:

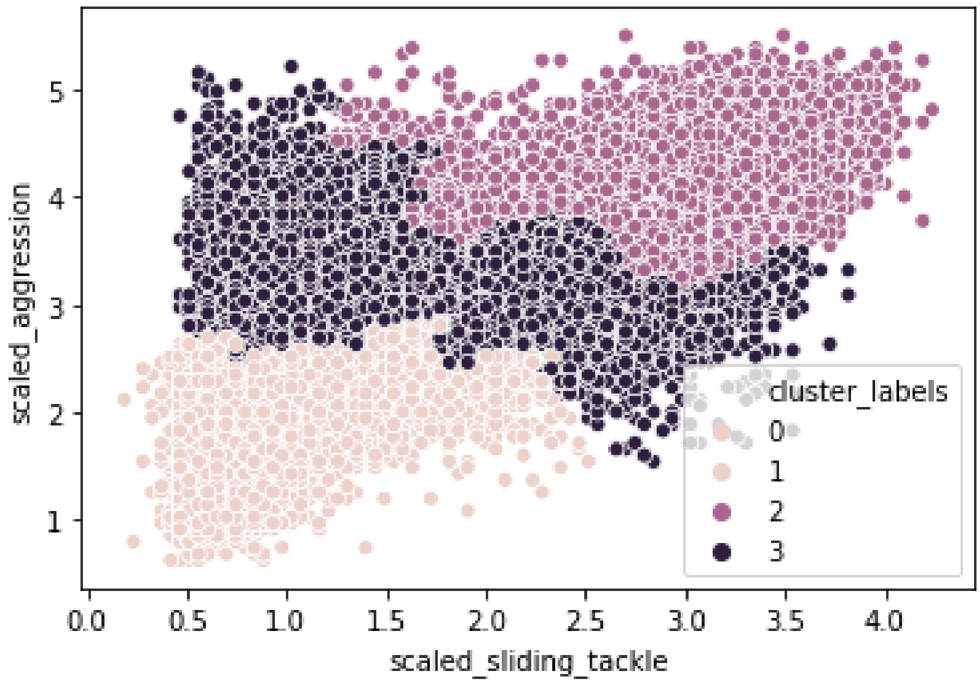
	sliding_tackle	aggression
0	23	63
1	26	48
2	33	56
3	38	78
4	11	29

[16]:
```
# Scaling sliding_tackle and aggression
Fifa['scaled_sliding_tackle'] = whiten(Fifa['sliding_tackle'])
Fifa['scaled_aggression'] = whiten(Fifa['aggression'])
```

[17]:
```
# Fitting the data into a hierarchical clustering algorithm
distance_matrix = linkage(Fifa[['scaled_sliding_tackle', 'scaled_aggression']],␣
↪'ward')
```

[9]:
```
# Fitting the data into a hierarchical clustering algorithm
distance_matrix = linkage(Fifa[['scaled_sliding_tackle', 'scaled_aggression']],␣
↪'ward')

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

# Displaying cluster centers of each cluster
print(Fifa[['scaled_sliding_tackle', 'scaled_aggression', 'cluster_labels']].
↪groupby('cluster_labels').mean())

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

	scaled_sliding_tackle	scaled_aggression
cluster_labels		
1	0.987373	1.849142
2	3.013487	4.063492
3	1.934455	3.210802



We can see that players are distributed in three clusters based on their scores on sliding_tackle and aggression.

Basics of K-Means clustering:

In K-Means clustering we generate cluster centres and assign the cluster labels .The cluster center is also known as the code book. The distortion is calculated as the sum of square of distances between the data points and cluster centers. Vq method is used to generate cluster labels.

K-means clustering:

Here I will use k-means clustering on a dataset. I will use the Comic Con dataset and check how k-means clustering works on it.

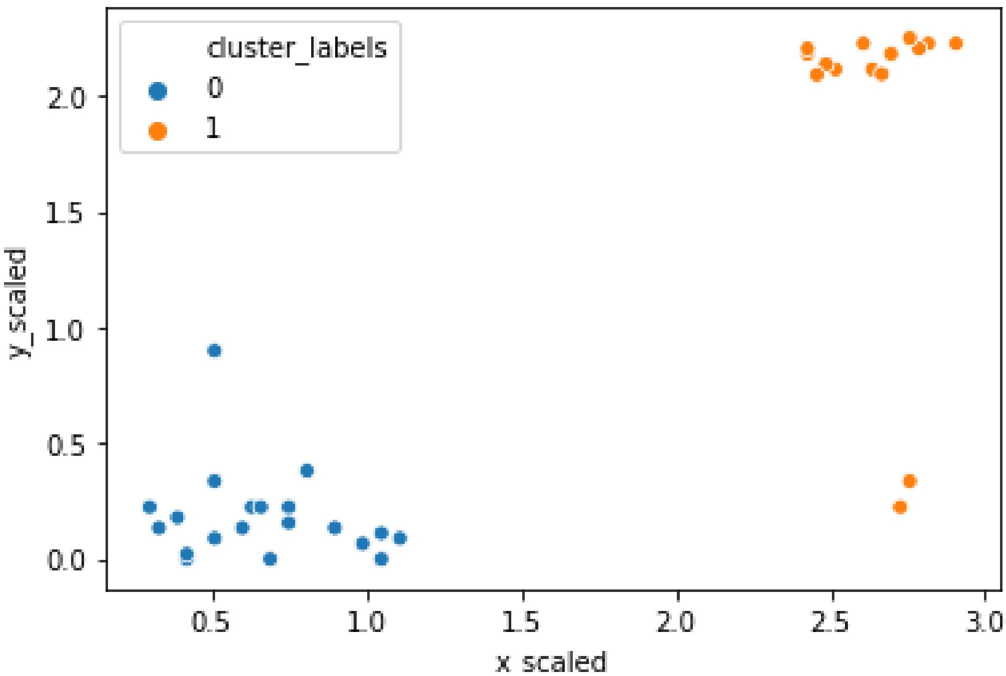
Recalling the two steps of k-means clustering:

Defining cluster centers through kmeans() function. It has two required arguments: observations and number of clusters. Assigning 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.

[85]:
```
# Importing the kmeans and vq functions
from scipy.cluster.vq import kmeans, vq

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

```
# Assigning cluster labels
comic_con['cluster_labels'], distortion_list = vq(comic_con[['x_scaled',␣
↪'y_scaled']],cluster_centers,distortion)

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



The clusters formed are exactly the same as hierarchical clustering.

How many clusters

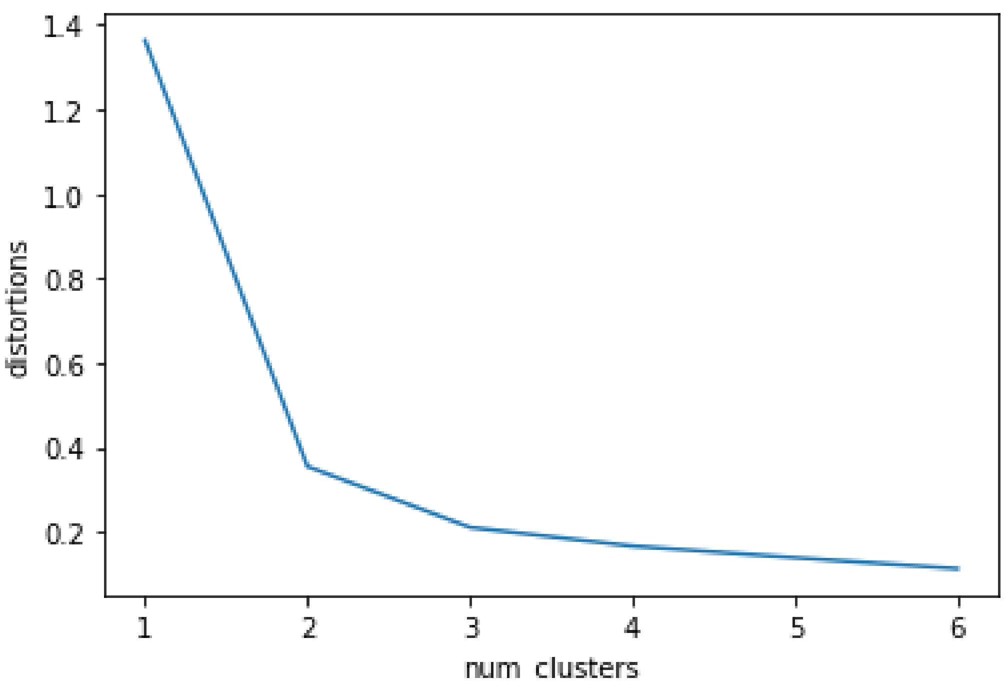
Distortion has an inverse relationship with the number of clusters which means that distortion decreases with increasing number of clusters. Segmenting the data into smaller fragments will lead to clusters being closer together, leading to a lower distortion. This is the underlying logic of the elbow method, which is a line plot between the number of clusters and their corresponding distortions.

The elbow method fails when the data is evenly distributed. There are other methods to find the ideal number of clusters such as the average silhouette and gap statistic methods.

Elbow method on distinct clusters

I will use the comic con data set to see how the elbow plot looks on a data set with distinct, well-defined clusters.

[87]:
```
distortions = []
num_clusters = range(1, 7)

# Creating 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)

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

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



From the elbow method we can see that the slop doesn't decrease after 3 clusters. In this case 3 is the right number of clusters.

Elbow method on uniform data

Above I 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.

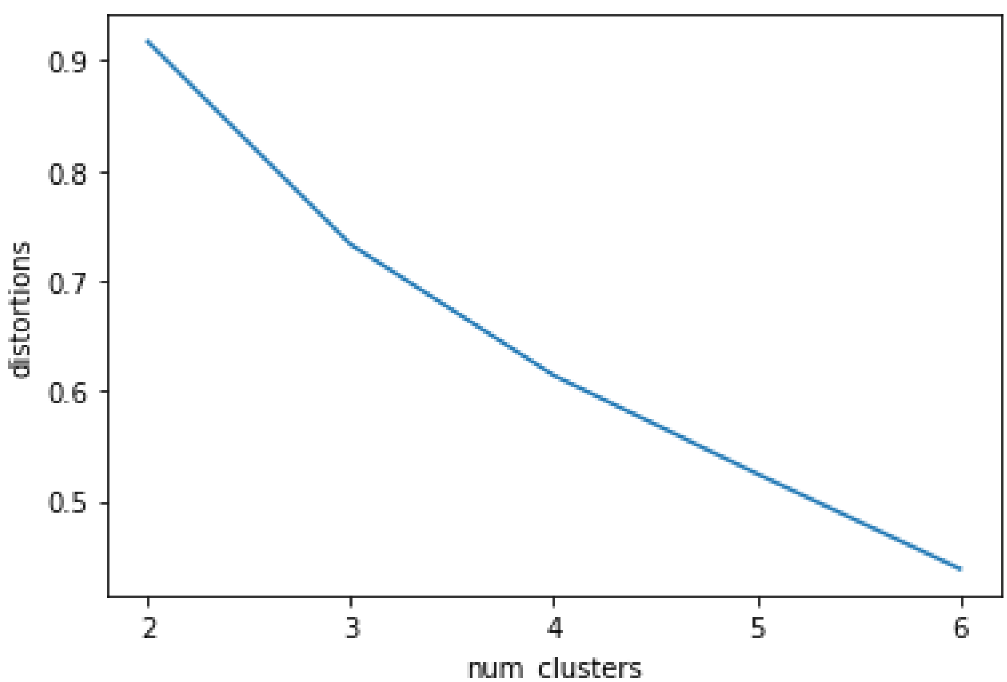
The data is stored in a Pandas data frame, uniform_data. x_scaled and y_scaled are the column names of the standardized X and Y coordinates of points.

Importing uniform data to try elbow method

[91]:
```
fp3 = '/Users/MuhammadBilal/Desktop/Data Camp/Cluster Analysis in Python/Data/
↪uniformdata.csv'
```

[92]:
```
uniform_data = pd.read_csv(fp3)
```

[93]:
```
distortions = []
num_clusters = range(2, 7)

# Creating 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)

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

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



There is no well defined elbow in this plot.

Limitations of K-means clustering

The first issue is the procedure to find the right number of clusters, k. The elbow method is one of the ways to determine the right k, but may not always work. The next limitation of k-means clustering is the impact of seeds on clustering. The final limitation is the formation of equal-sized clusters.

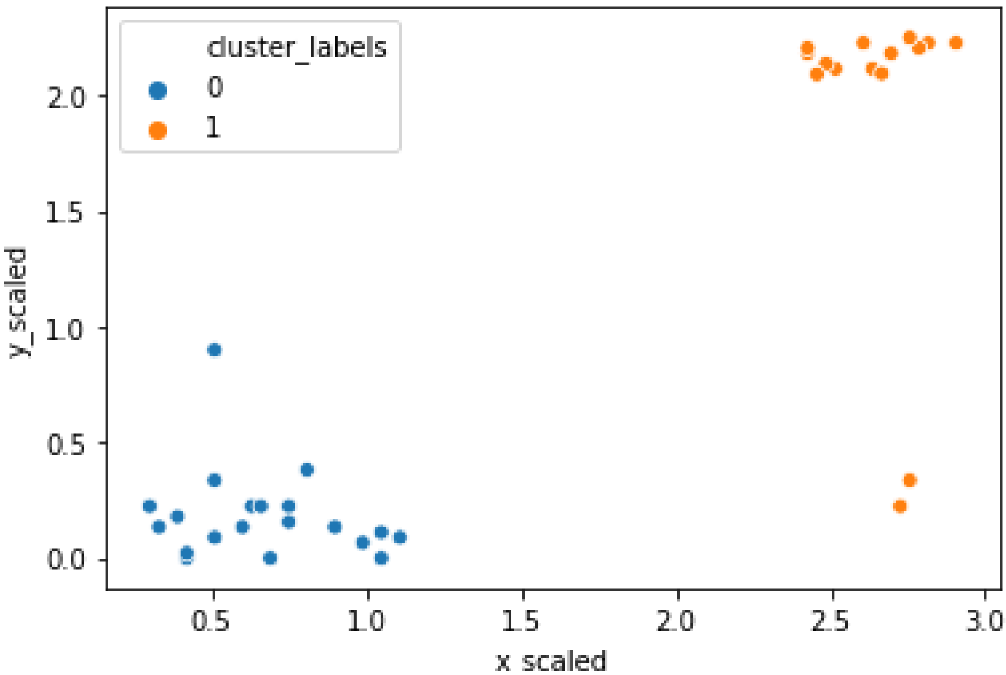
As the process of defining the initial clusters is random, the initialization can affect the final clusters. To get consistent results when running k-means clustering on the same dataset multiple times, it is a good idea to set the initialization parameters for random numbers generations.

The effect of seeds is only seen when the data to be clustered is fairly uniform. If the data has distinct clusters before clustering is performed, the effect of seeds will not result in any changes in the formation of resulting clusters.

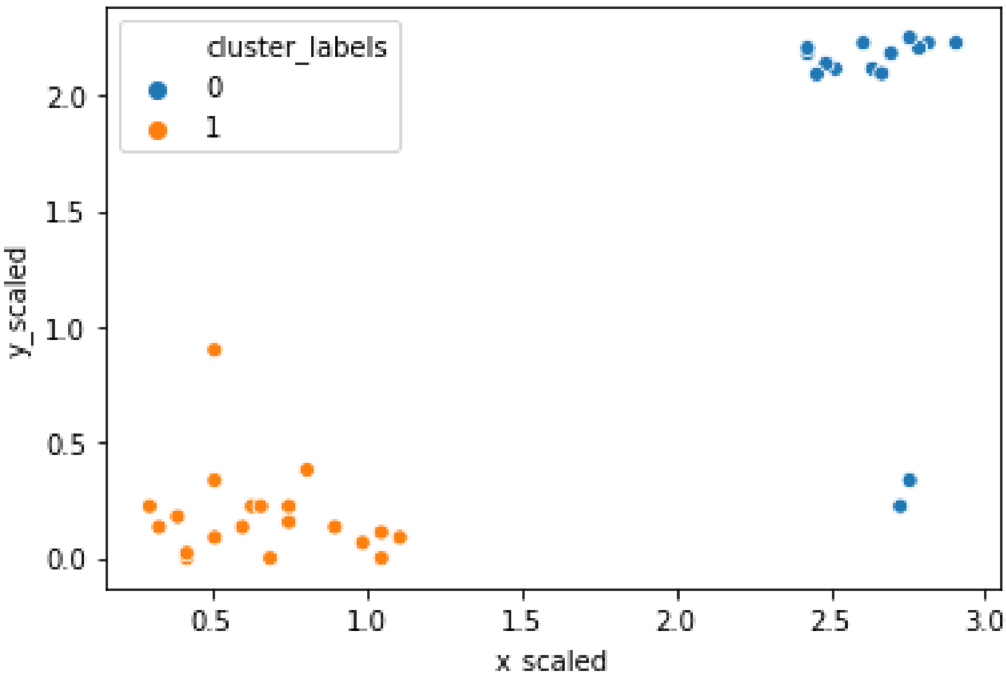
Impact of seeds on distinct clusters

We noticed the impact of seeds on a dataset that did not have well-defined groups of clusters. Here I will explore whether seeds impact the clusters in the Comic Con data, where the clusters are well-defined.

[95]:
```
# Importing random class
from numpy import random

# Initializing seed
random.seed(0)

# Running 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)

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



[96]:
```
# Importing random class
from numpy import random

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

# Running 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)

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



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

FIFA 18: defenders revisited

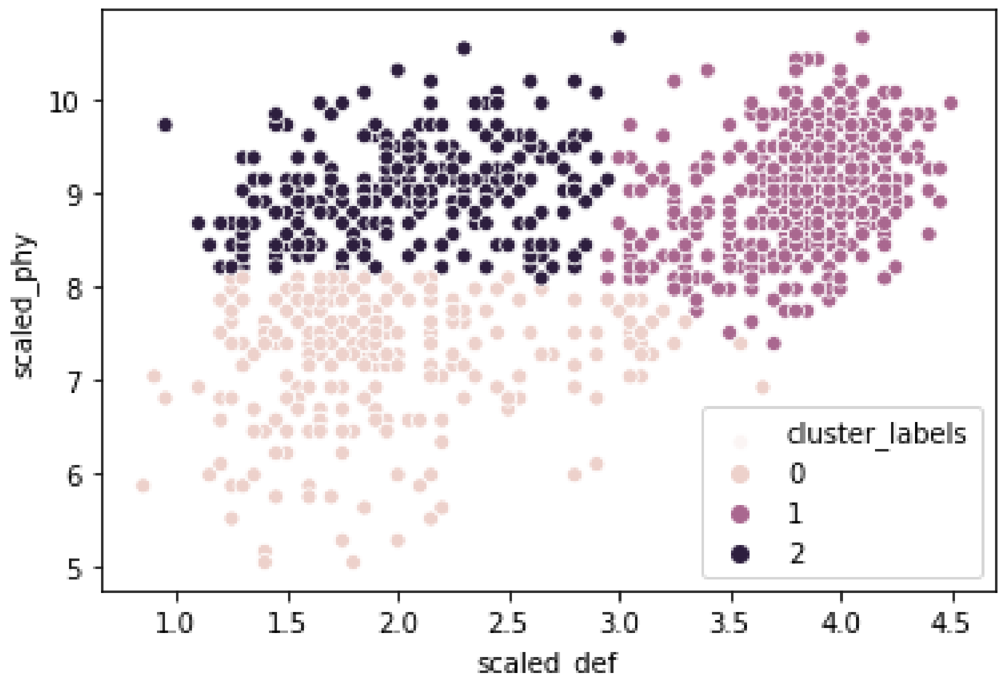
In the FIFA 18 dataset, various attributes of players are present. Two such attributes are:

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.

[98]:
```
# 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()
```

```
                scaled_def  scaled_phy
cluster_labels
0                 1.948298    7.163234
1                 3.817844    9.020452
2                 2.072803    9.066327
```



[ ]:
```
The seed has an impact on clustering as the data is uniformly distributed.
```

Document clustering

One of the uses of unsupervised learning techniques is to group items such as news together by a service such as Google News. This technique is known as document clustering. Document clustering uses some concepts from natural language processing or NLP. First we clean the data that does not add value to the analysis. We remove some items from the data that include punctuation, emoticons, and words such as "the", "is", "are" etc. Next we find the TF-IDF of the terms or a weighted statistic that describes the importance of a term in a document. Finally we cluster the TF-IDF matrix and display the top terms in each cluster. The text cannot be analyzed before converting into smaller parts called tokens which we achieve by using NLTK's word-tokenize method. First we remove all special characters from tokens and check if it contains any stop words. Finally we return the cleaned tokens. Once relevant terms have been extracted, a matrix is formed with the terms and documents as dimensions. An element of the matrix signifies how many times a term has occurred in each document. Most elements are zeros, hence, sparse matrices are used to store these matrices more efficiently. A sparse matrix only contains terms which have non-zero elements. To find the TF-IDF of terms in a group of documents we use the TfidfVectorizer class of sklearn. The

TF-IDF matrix returns a sparse matrix. K-menas in scipy does not work with sparse matrices, so we convert the tfidf matrix to its expanded form using the todense method. K-means can then be applied to get the cluster centers. Here we cannot use the elbow plot, as it will take an erratic form due to the high number of variables. Each cluster center is a list of tfidf weights, which signifies the importance of each term in the matrix. To find the top terms we first create a list of all terms. We can also modify the remove_noise method to filter hyperlinks or replace emoticons with text. We can also normalize every word to its base form: for instance, run, ran and running are the forms of the same verb run. The todense method may not work with large datasets and we may need to consider an implementation of k-means that works with sparse matrices.

TF-IDF of movie plots

I will 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.

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

[172]:
```
from nltk.tokenize import word_tokenize
import re

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
```

[173]:
```
# Importing TfidfVectorizer class from sklearn
from sklearn.feature_extraction.text import TfidfVectorizer

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

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

Top terms in movie clusters

Now that I have created a sparse matrix, I will generate cluster centers and print the top three terms in each cluster. Using the .todense() method to convert the sparse matrix, tfidf_matrix to a normal matrix for the kmeans() function to process. Then, I will 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 here.

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.

[174]:
```
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])
```

```
['her', 'she', 'him']
['him', 'they', 'an']
```

Clustering with multiple features:

This time I will take more than 2 features of the fifa dataset and try to interpret and validate the results of clustering. It is important to understand that all features can bot be visualized and assessed, at the same time when clustering with more than 3 features we can validate the results. This step assumes that we have created the elbow plot, performed the clustering process and generated cluster labels. First we can check how the cluster centers vary with respect to the overall data. if we notice that cluster centers of some features do not vary significantly with respect to the overall data, it is an indication that we can drop that feature in the next run. We can also look at the sizes of the clusters formed. If one or more clusters are significantly smaller than the rest we can double check if their cluster centers are similar to the other clusters. If the answer is yes, we can reduce the number of clusters in subsequent runs. It is because I have performed clustering on three attacking attributes on the fifa dataset for which goalkeepers have a very low value as indicated by the cluster centers. The smaller cluster is composed primarily of goalkeepers as I will explore later. Even though all variables cannot be visualized across clusters, there are other simpler visualizations that can help to understand the results of clustering. We may visualize cluster centers or other variables stacked against each other.

In pandas, we can use the plot method after groupby to generate such plots. Bar chart demonstrates this trend. We can also create a line chart to see how variables vary across clusters. In case of fifa we will see that all three attributes are significantly higher in one cluster.

When dealing with a large number of features, certain techniques of feature reduction may be used. Two popular tools to reduce the number of features are factor analysis and multidimensional scalling.

Basic checks on clusters

In the FIFA 18 dataset, we have concentrated on defenders in previous exercises. Here I will 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 cluster labels are stored in the cluster_labels column. Recalling the .count() and .mean() methods in Pandas help to find the number of observations and mean of observations in a data frame.

[185]:
```
# 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())
```

```
cluster_labels
0    242
1    501
2    257
Name: ID, dtype: int64
cluster_labels
0    67971.074380
1    69043.912176
2    71564.202335
Name: eur_wage, dtype: float64
```

In this example, the cluster sizes for two clusters are almost identical, and there are no significant differences that can be seen in the wages. Further analysis is required to validate these clusters.

[11]:
```
FIFA = fifa[['ID', 'name', 'scaled_def','scaled_phy']]
```

[12]:
```
FIFA.head()
```

[12]:

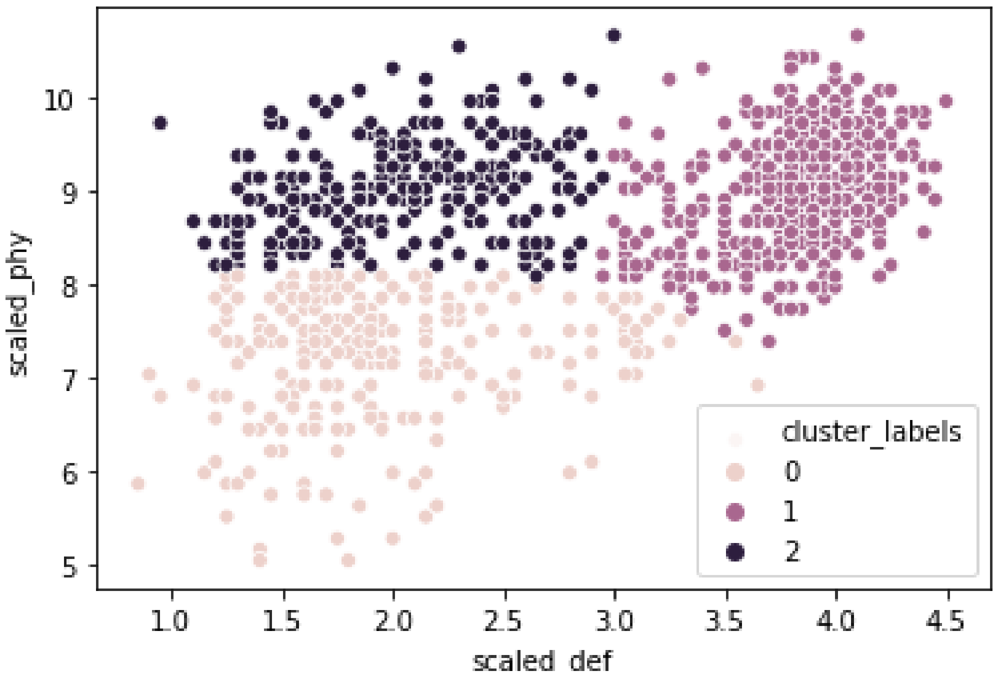
	ID	name	scaled_def	scaled_phy
0	20801	Cristiano Ronaldo	1.649258	9.374085
1	158023	L. Messi	1.299416	7.147740
2	190871	Neymar	1.499326	7.030564
3	176580	L. Suárez	2.099056	9.491261
4	167495	M. Neuer	2.998652	10.663022

[27]:
```
from numpy import random

# 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()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  # Remove the CWD from sys.path while we load stuff.
```

```
                scaled_def  scaled_phy
cluster_labels
0                 1.948298    7.163234
1                 3.817844    9.020452
2                 2.072803    9.066327
```



[34]:
```
scaled_features = ['scaled_pac', 'scaled_dri', 'scaled_sho']
```

[35]:
```
fifa['scaled_pac'] = whiten(fifa['pac'])
fifa['scaled_dri'] = whiten(fifa['dri'])
fifa['scaled_sho'] = whiten(fifa['sho'])
```

[36]:
```
# Fitting the data into a k-means algorithm
cluster_centers,_ = kmeans(fifa[['scaled_pac', 'scaled_dri', 'scaled_sho']], 3)

# Assigning cluster labels
fifa['cluster_labels'], _ = vq(fifa[['scaled_pac', 'scaled_dri',␣
↪'scaled_sho']], cluster_centers)

# Displaying cluster centers
print(fifa[['scaled_pac', 'scaled_dri', 'scaled_sho', 'cluster_labels']].
↪groupby('cluster_labels').mean())
```

```
                scaled_pac  scaled_dri  scaled_sho
cluster_labels
0                 5.916146    7.999613    5.040766
1                 5.290410    6.318656    3.229186
2                 7.081310    8.710852    5.520627
```

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). In below codes, I will use all six characteristics to create clusters.

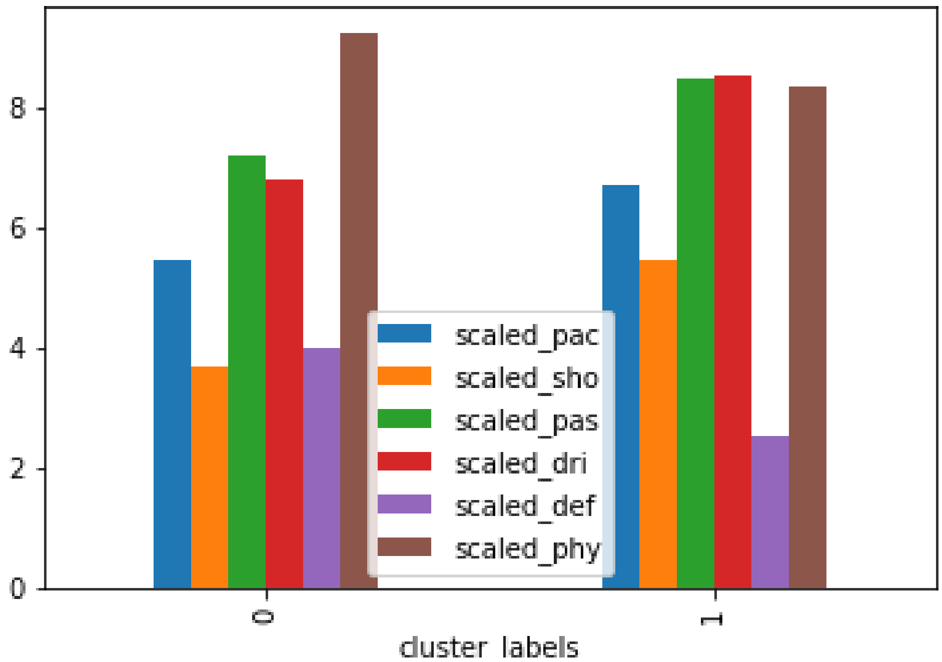
[38]:
```
Scaled_features =␣
↪['scaled_pac','scaled_sho','scaled_pas','scaled_dri','scaled_def','scaled_phy']
print(fifa[['scaled_pac','scaled_sho','scaled_pas','scaled_dri','scaled_def','scaled_phy']].
↪describe())
```

```
        scaled_pac   scaled_sho   scaled_pas   scaled_dri   scaled_def  \
count  1000.000000  1000.000000  1000.000000  1000.000000  1000.000000
mean      6.334742     4.930355     8.091719     8.018715     2.916938
std       1.000500     1.000500     1.000500     1.000500     1.000500
min       2.323034     1.387146     4.079401     4.237661     0.849618
25%       5.850604     4.453469     7.592219     7.627791     1.949124
50%       6.452873     5.256553     8.272119     8.263440     3.098607
75%       7.055141     5.621592     8.725386     8.687206     3.898247
max       8.259677     6.789714    10.765086    10.170387     4.497978

        scaled_phy
count  1000.000000
mean      8.582795
std       1.000500
min       5.038571
25%       7.967972
50%       8.788205
75%       9.256909
max      10.663022
```

[39]:
```
# Creating centroids with kmeans for 2 clusters
cluster_centers,_ = kmeans(fifa[Scaled_features], 2)

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

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

# Getting 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])
```

```
                scaled_pac  scaled_sho  scaled_pas  scaled_dri  scaled_def  \
cluster_labels
0                 5.464781    3.677603    7.185542    6.780258     3.96738
1                 6.684923    5.434618    8.456477    8.517224     2.49411
```

```
                 scaled_phy
cluster_labels
0                  9.208732
1                  8.330840
```




```
1 ['Cristiano Ronaldo' 'L. Messi' 'Neymar' 'L. Suárez' 'M. Neuer']
0 ['Sergio Ramos' 'G. Chiellini' 'L. Bonucci' 'J. Boateng' 'D. Godín']
```

That is correct! 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.

```
[ ]:
```