Cluster Analysis in Python
Course by Shaumik Daityari and datacamp

Clustering is an unsupervised learning algorithm
example
scans through the text of each article and based on frequently occurring terms, groups articles together

Labeled and unlabeled data
example
no labels - just coordinates
labeled - coordinates are associated with a group

What is unsupervised learning?
an umbrella term for a group of machine learning algorithms that are used to find patterns
the data in these algorithms is not labeled, classified, or characterized prior to running the algorithm
the goal of the algorithm is to find inherent structure within the data
Common unsupervised learning algorithms:
clustering
anomaly detections
neural networks
clustering is used to group similar data points together

Clustering algorithms
hierachical
K means
other less common > DBSCAN, Gaussian Methods

Hierachical clustering
the first step, all points are considered as individual clusters
a cluster center is a mean of attributes of all data points in a cluster
example
13 x and y coordintates
in this case cluster centers will have two attributes > the mean of x and the mean of y
at this point, cluster centers of all clusters are the coordinates of the individual points
next, the distance between all pairs of cluster centers are computed and the two closest clusters are merged

the cluster center of the merged cluster is then recomputed
now we have 12 clusters
second step, repeat
the clusters with the closest cluster centers are merged
at every step the number of clusters reduces by one
we continue until we arrive at the desired clusters
example

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

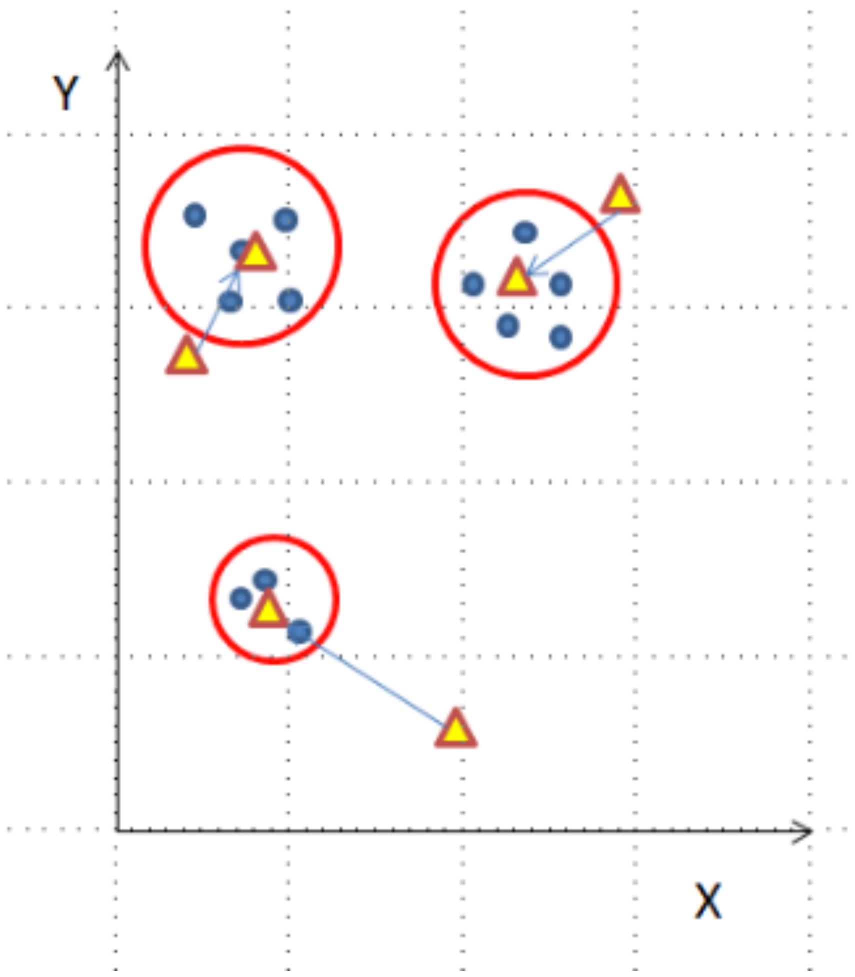
```
x_coordinates = [80.1, 93.1, 86.6, 98.5, 86.4, 9.5, 15.2, 3.4,
                 10.4, 20.3, 44.2, 56.8, 49.2, 62.5, 44.0]
y_coordinates = [87.2, 96.1, 95.6, 92.4, 92.4, 57.7, 49.4,
                 47.3, 59.1, 55.5, 25.6, 2.1, 10.9, 24.1, 10.3]

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

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

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

we need to use scipy
linkage method computes distances bewtween intermediate clusters
fcluster method generates clusters and assigns associated cluster labels to a new column in the DataFrame
use the hue argument to associate clusters with different colors

K-means clustering
example using 13 points with x, y coordinates
first, a random cluster center is generated for each of the three clusters
next, the distance to these cluster centers is computed for each point to assign to the closest center
then, the cluster centers are recomputed
this iteration of assigning points to the recomputed cluster centers is performed a predefined number of times
to visualize:



the outside triangle is the first random generated clusters
the triangle within is the recomputed cluster center
Python example

```
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)
```

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

using scipy
centroids of the clusters are computed using kmeans and cluster assignments for each point are done through vq
the second argument in both methods is distortion
distortion is captured in a dummy variable

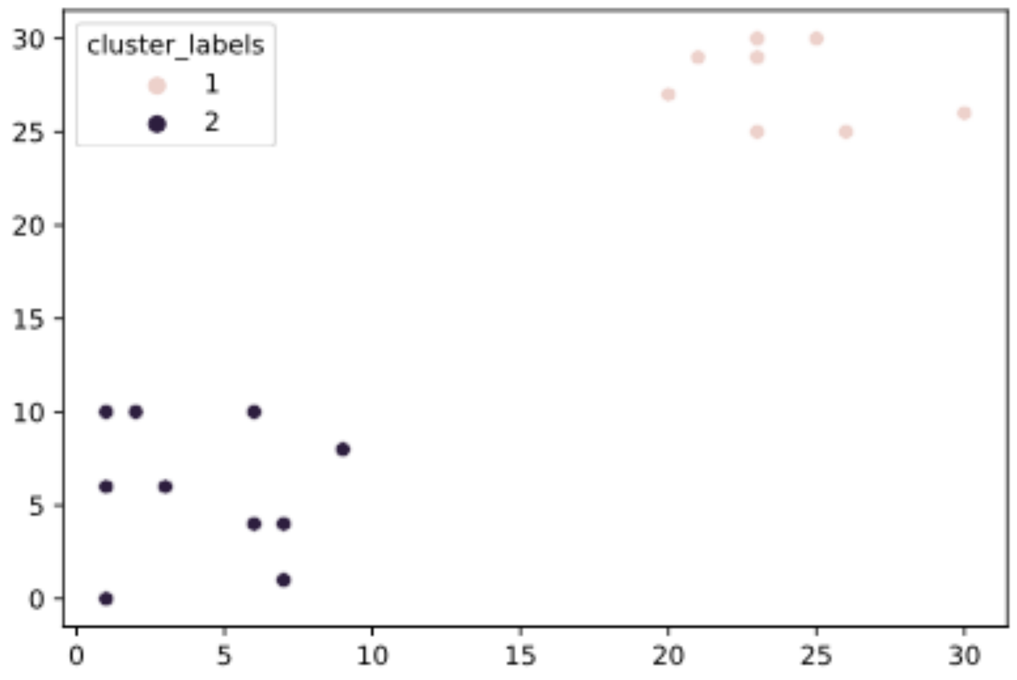
Example
# Import linkage and fcluster functions
from scipy.cluster.hierarchy import linkage, fcluster

# Use the linkage() function to compute distance
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()

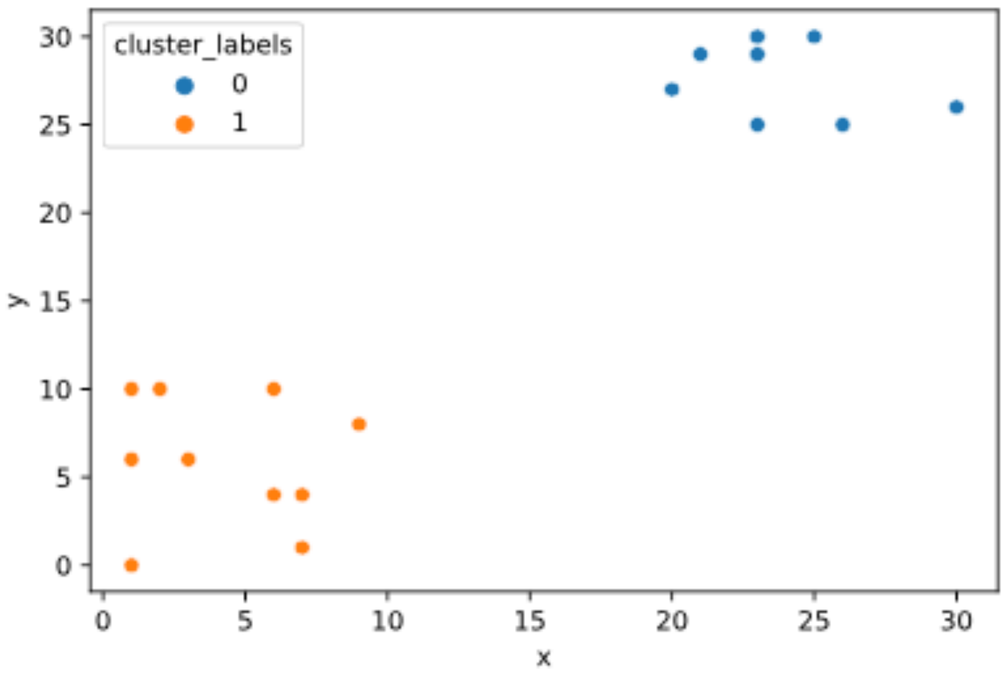
output>



Example

```
# 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()
```

output>



Data preparation for cluster analysis
why?
variables may have incomparable units
variables may be same unit but have vastly different scales and variances
all these things (ie data in its raw form) may lead to bias in clustering
how?
clusters formed may be dependent on one variable significantly more than the other
how do we get around these issues?
by using normalization of individual variables

Normalization is a process by which we rescale the values of a variable with respect to standard deviation of the data
the resulting standard deviation post normalization is 1
the process is simple
divide the value (we will call it x) by its standard deviation
x_new = x / std_dev(x)
Python example

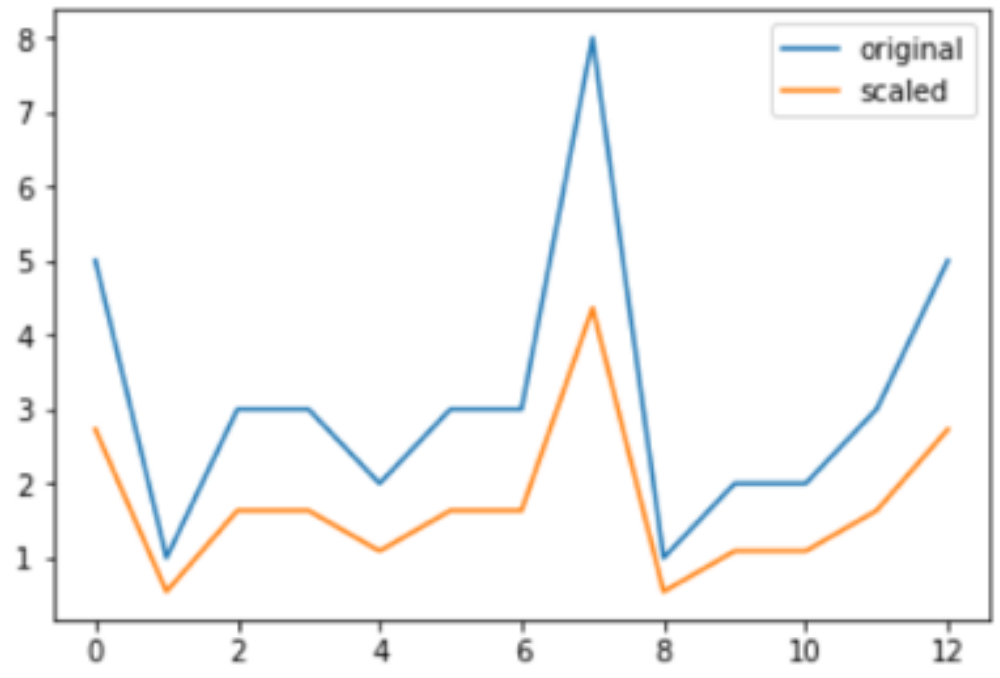
```
from scipy.cluster.vq import whiten
```

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

```
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]
```

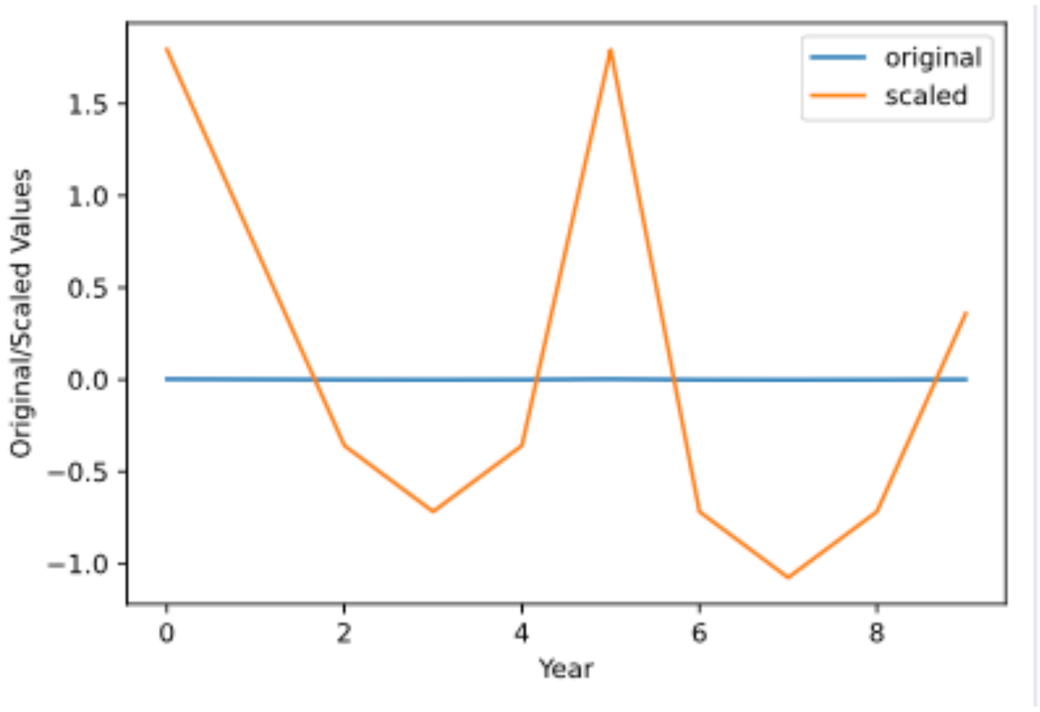
we use the Whiten method
data should be in a list
the data can be multi-dimensional
if multi-dimensional, Whiten method divides each value by the standard deviation of the column
the output of the Whiten method is an array of the same dimensions
how to visualize normalization with pyplot
#need to plot the original and scaled data
#default pyplot plots line graphs
plt.plot(data, label='original')
plt.plot(scaled_data, label='scaled')
plt.legend()
plt.show()
output>



the serial number of the points is in the x axis
the value of the original and scaled data points is in the y axis

Example
```
# 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()
```

output>



notice how the changes in the original data are negligible compared to the scaled data

Basics of hierarchical clustering

the critical step is to compute the distance matrix at each stage
we do this with the linkage method
this process computes the distances bewteen clusters as we go from N clusters to 1 cluster, where N is the number of original points
there are four parameters or arguments for this method
scipy.cluster.hierarchy.linkage(observations, method='single', metric='euclidean', optimal_ordering=False)

1. observations
2. method - tells the algorithm how to calculate the proximity between two clusters
    1. 'single' - decides the proximity of clusters based on their two closest objects
    2. 'complete' - decides the proximity of cluster centers based on their two farthest objects
    3. 'average' - based on the arithmetic mean of all objects
    4. 'centroid' - based on geometric mean of all objects
    5. 'median' - based on the median of all objects
    6. 'ward' - computes cluster proximity using the difference bewtween summed squares of their joint clusters minus the individual summed squares (*focuses on clusters more concentric towards its center)
3. metric - decides the distance between two objects
    1. common 'euclidean' - the distance is a straight line between two points on a 2D plane
    2. you can also put your own function here
4. optimal_ordering - optional argument, that changes the order of linkage matrix

Next, fcluster method
scipy.cluster.hierarchy.fcluster(distance_matrix, num_clusters, criterion)

1. distance_matrix - output of linkage() method
2. num_clusters - number of clusters
3. criterion - how to decide thresholds to form clusters (commonly use 'maxclust')

**remember like everything there is no right method for all
need to carefully understand the distribution of data and you work from there

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

# 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()
```

Visualizing clusters
can easily see trends
often we will add a separate column for cluster centers
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

Example - using just matplotlib

```
# 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()
```

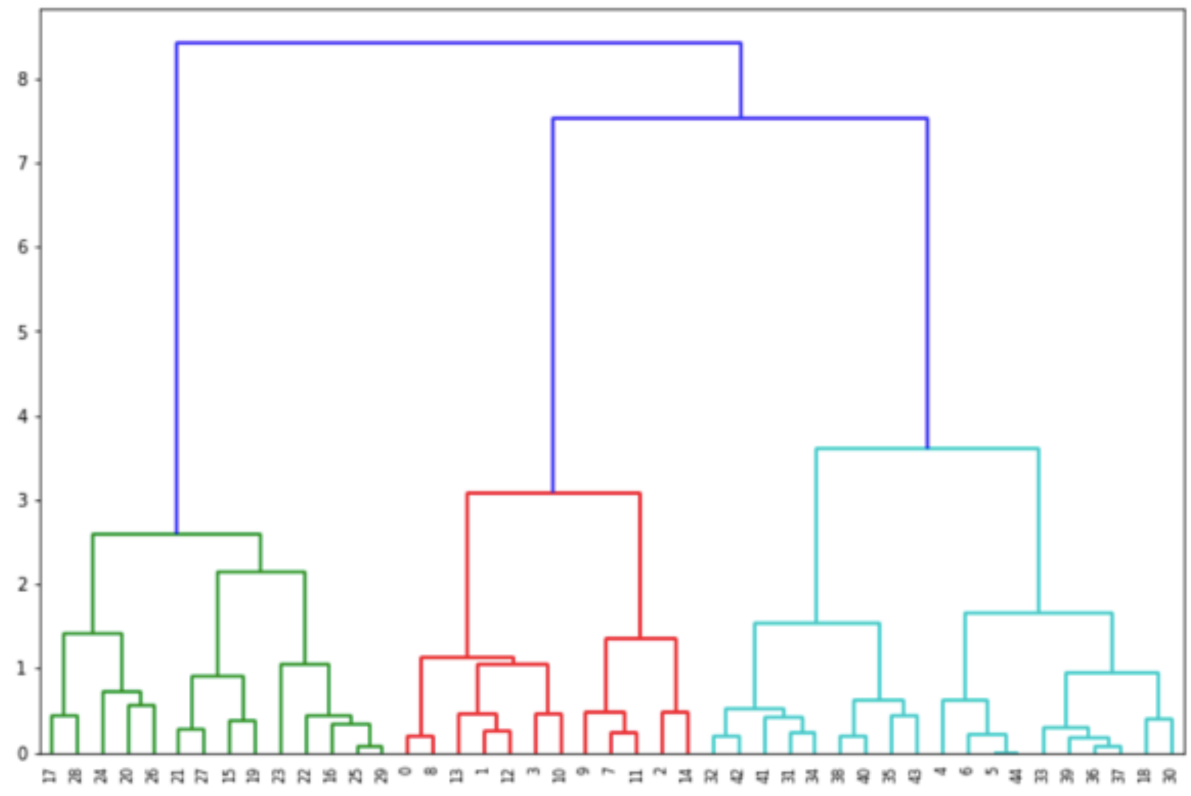
Example - using the preferred way with seaborn

```
# 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()
```

How to decide how many clusters?
can graphically look at the number of points in our dataset
in hierarchical clustering we can use a graphical diagram called a dendrogram
a dendrogram is a branching diagram that shows the progression in a linkage

object as we proceed through the hierarchical clustering algorithm
example - using a dendrogram

```
from scipy.cluster.hierarchy import dendrogram
Z - linkage(df[['x_whiten;, 'y_whiten']], method='ward', metric='euclidean')
dn = dendrogram(Z)
plt.show()
```



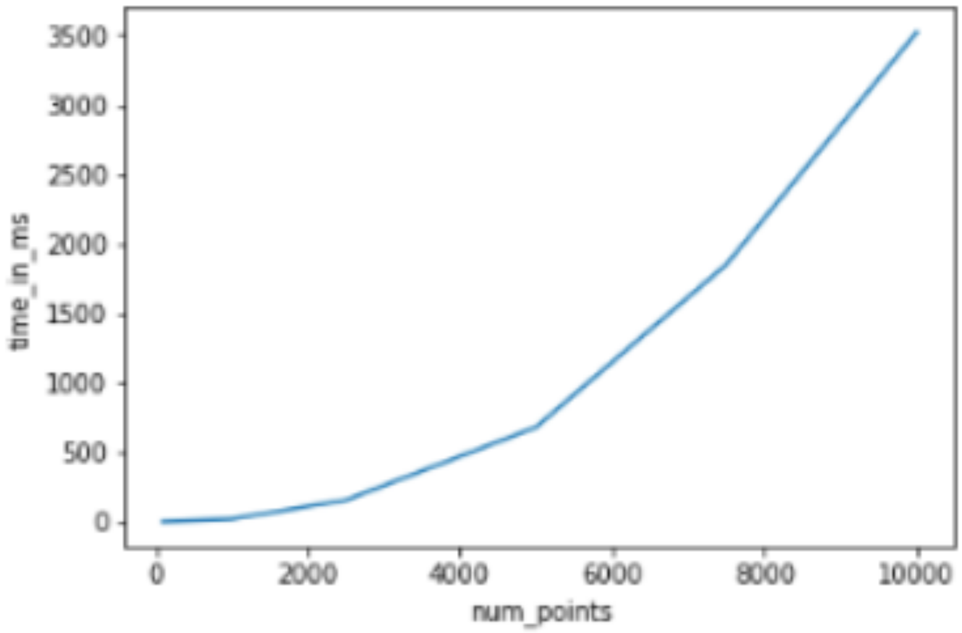
to understand
the hierarchical clustering algorithm - each step was a result of merging of two closest clusters in the earlier step
the x axis represents individual points
the y axis represents the distance or dissimilarity between clusters
each inverted U represents a cluster divided into its two child clusters
the inverted U at the top of the figure represents a single cluster of all the data points
the width of the U shape represents the distance between the two child clusters
a wider U means that the two child clusters were farther away from each other as compared to a narrower U in the diagram
*if you say draw a horizontal line starting at the 5 on the y-axis
this line can be drawn at any stage
the y axis represents the stages
*the number of vertical lines that the horizontal line intersects tells you the number of clusters at that stage

the distance between those vertical lines indicates the inter-cluster distance
in our example with a horizontal line at 5, we would have three clusters
how many clusters to choose still depends on the problem and the data but this process helps
visualizing a scatterplot in conjunction with the dendrogram can help drive your decision

Limitations of hierarchical clustering
by using the timeit module we can see that the increase of points with linkage does not create a linear relationship
instead it is quadratic
we can test it with this code

```
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')
```

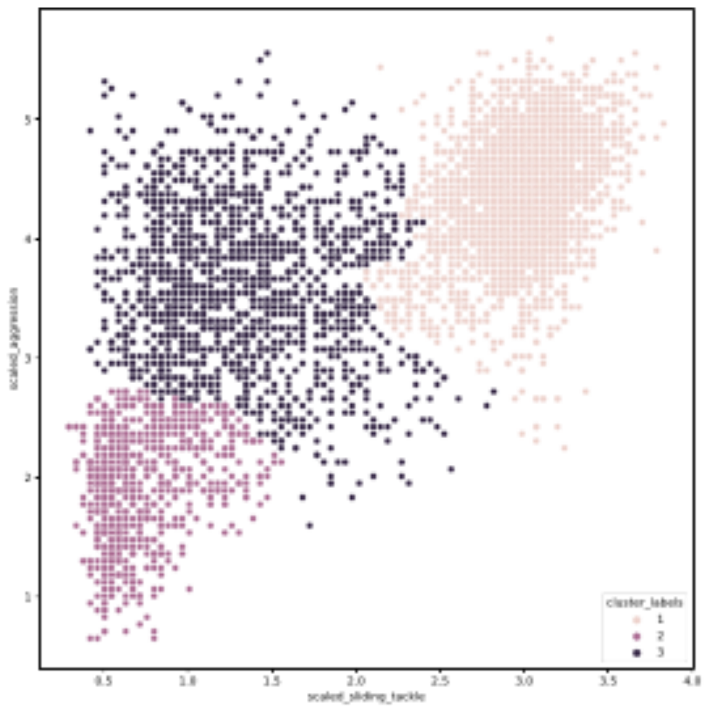
plotting out progressively larger datasets shows this



this means that the technique of hierarchical clustering becomes infeasible for huge numbers of data points

Example
```
# 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()
```

output>



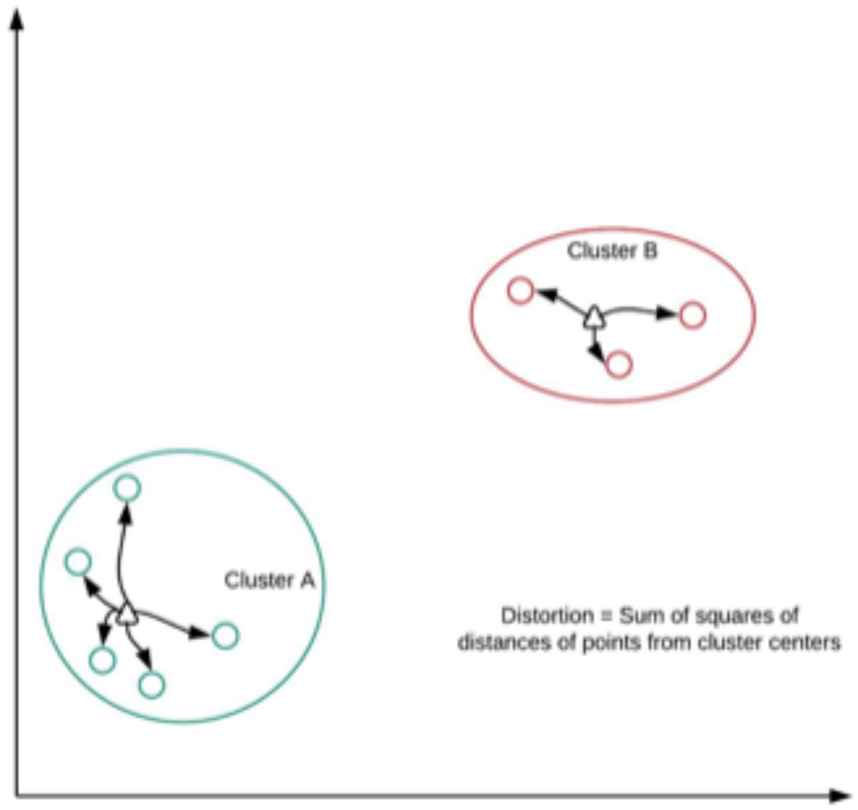
Basics of k-means clustering
not limited by runtime like hierarchical clustering
allows you to cluster large datasets in a fraction of the time

Step 1
kmeans() method accessed through scipy

5 arguments

1. obs - list of standardized observations (standardized using the whiten() method)
2. k_or_guess - the number of clusters
3. iter - number of iterations of the algorithm to perform (default is 20)
4. thres - threshold, the idea behind this argument is that the algorithm is terminated if the change in distortion since the last k-means iteration is less than or equal to the threshold (default is 1e-05 or 0.00001
5. check_finite - a boolean value indicating if a check needs to be performed on the data for the presence of infinite or naN values (default is True), this ensures that data points with NaN or infinite values are not considered for classification, which ensures that the results are accurate and unbiased

kmeans function returns two arguments > cluster centers and distortion
cluster centers is also known as the code book
distortion is calculated as the sum of square of distances between the data points and cluster centers, as demonstrated in this figure:


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

Step 2
use the vq method to generate cluster labels
the vq method takes three arguments

1. obs - standardized observations through the whiten() method
2. code_book - is the first output of the kmeans() method

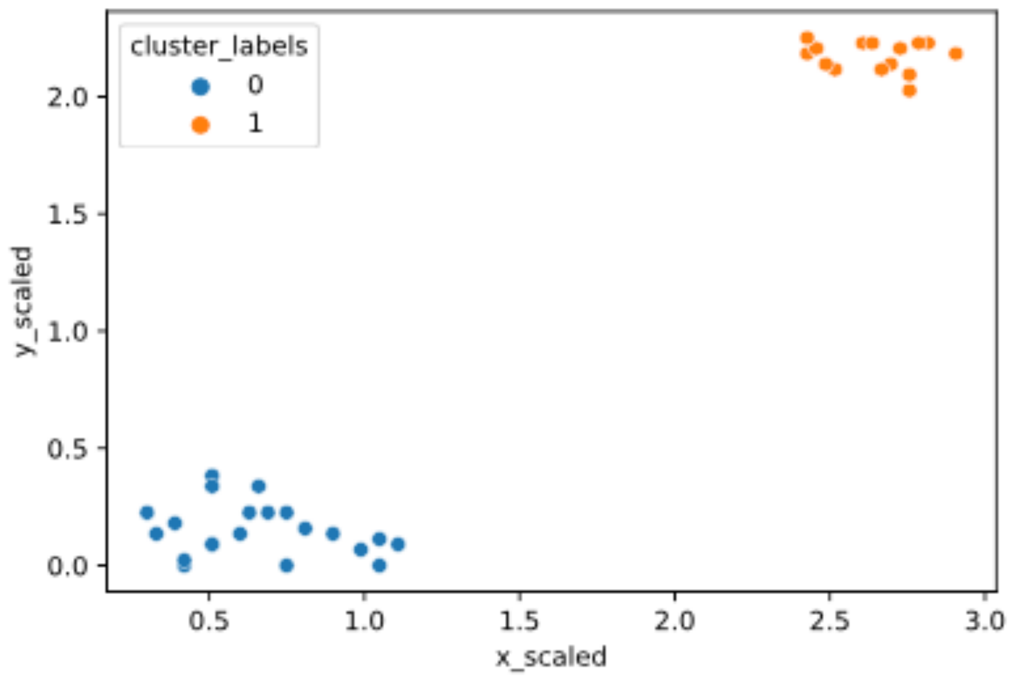
3. check_finite - checks for NaNs and infinity(default is True)

returns cluster labels (also known as the ‘code book index’) and the distortion

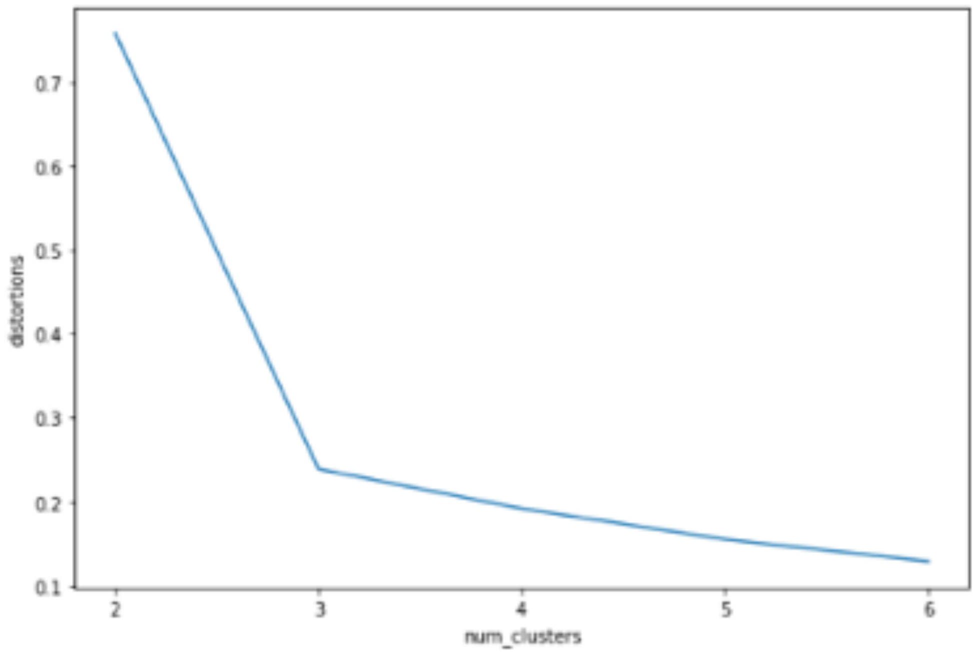
Distortions
kmeans() returns a single value of distortions based on the overall data
vq() returns a list of distortions, one for each data point
*the mean of the list of distortions from the vq method should approximately equal the distortion value of the kmeans method if the same list of observations is passed

Example
```
# 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()
```

ouput>



How to find the right k?
no absolute method to find right number of clusters (k) in k-means clustering
elbow method/plot can help you decide



Distortion has an inverse relationship with the number of clusters
this means that distortion decreases with increasing number of clusters

remember distortion is the sum of the squares of distances between each data point and its cluster center
more clusters means more and smaller fragments closer together which will lead to a lower distortion
distortion becomes zero when the number of clusters equals the number of points
this is the underlying logic of the elbow method
a line plot between the number of clusters and their corresponding distortions

Elbow method
first run kmeans with varying clusters
number of clusters on x-axis and distortion on the y-axis
the goal is to find the point at which the distortion decrease relatively less on increasing the number of clusters
Python example

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

num_clusters = range(2, 7)
```

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

```python
# 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()
```

the elbow plot above would tell us the ideal point is approximately at 3 clusters
*be aware the elbow method is not perfect
can fail when data is evenly distributed
other methods to find k > average silhouette and gap statistic

Example
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 DataFrame 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()
```

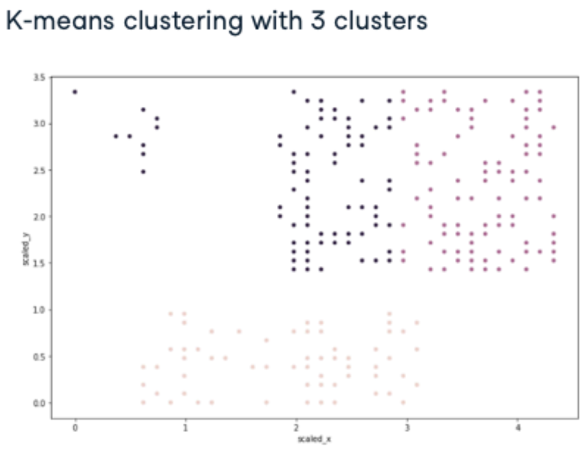
Limitations of k-means
how to find the right number of clusters
impact of seeds
biased towards equal sized clusters

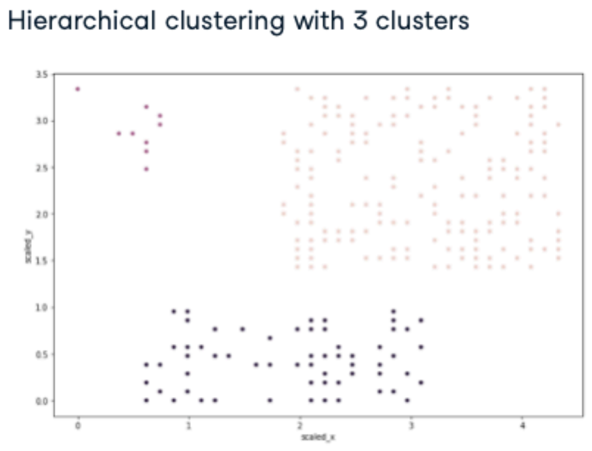
impact of seeds
the process of defining the initial cluster centers is random
so to get consistent results when running k-means on the same dataset multiple times it is imperative to use the seed method of random class in numpy

```
#initialize a random seed
from numpy import random
random.seed(12)
```

Test
passing two different seeds
the two cases have different clusters
many points along the boundaries are interchanged
interestingly, this effect of seeds is only seen when the data to be clustered is fairly uniform
if the data has distinct clusters before the clustering is performed, the effect of seeds will not result in any changes in the formation of resulting clusters

Comparing kmeans and hierarchical clustering with 3 clusters (group of 200, 70, and 10)





kmeans clusters in a non-intuitive way
this is because kmeans goal is to minimize distortions

Example

```
# 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()
```

Dominant colors in images
background
image consists of pixels
pixels consists of three values
each value is a number 0 to 255
the combo of these 3 numbers creates the color of the pixel
tools
matplotlib.image.imread > converts image to pixels
converts a jpeg into a matrix which contains the RGB values of each pixel
matplotlib.pyplot.imshow > displays colors of cluster centers once you perform

kmeans on the RGB values

Convert image to RGB matrix
example - using a jpeg image which is half sky and half sea

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

output > MxNx3 matrix where M and N are the dimensions of the image

```
#extract and store all RGB values and store them in their corresponding lists
r = [ ]
g = [ ]
b = [ ]

for row in image:
        for pixel in row:
        temp_r, temp_g, temp_b = pixel
        r.append(temp_r)
        g.append(temp_g)
        b.append(temp_b)
#once lists are created, store them in a pandas DataFrame
pixels = pd.DataFrame({‘red’: r, ‘blue’: b, ‘green’: g})
#view pixels
pixels.head()
#create an elbow plot
distortions = [ ]
num_clusters = range(, 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 DF with two lists - number of clusters and distortions
elbow_plot - pd.DataFrame({‘num_clusters’:num_clusters, ‘distortions’:distortions})
sns.lineplot(x=‘num_clusters’, y=‘distortions’, data=elbow_plot)
plt.xticks(num_clusters)
plt.show()
```

output> plot indicates two clusters which is agreeable with the visualization (color for the sky and color for the sea)

The cluster centers obtained are standardized RGB values
Again, standardized value of a variable is its actual value divided by the standard deviation
imshow() takes RGB values that have been scaled to the range of 0 to 1

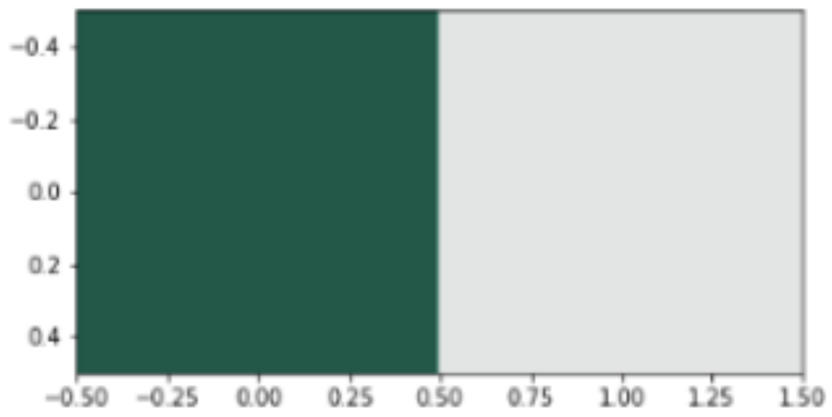
to do this we need to multiply the standardized values of the cluster centers with their corresponding standard deviations
maximum vlaue of RGB values is 255
so we divide by 255 to get a scaled value in the range of 0 to 1
example - continuation from above (sky/sea jpeg)

```
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 to 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))
```

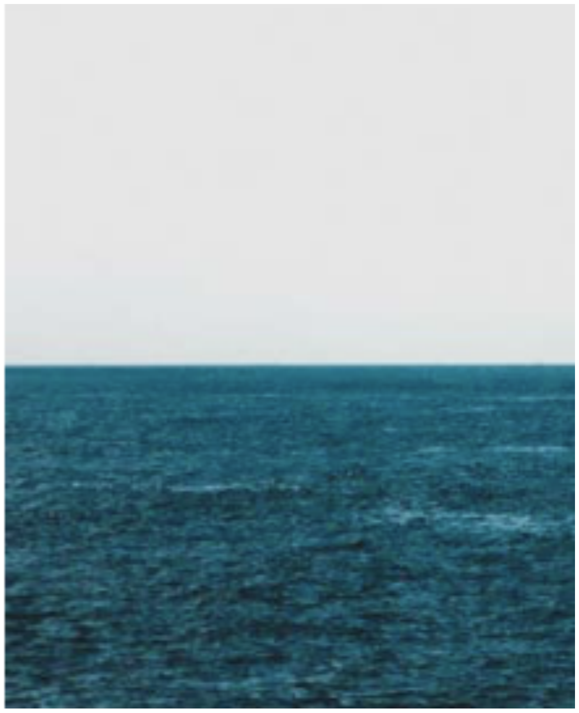
*need to provide the colors variable encapsulated as a list
imshow() expects a MxN.3 matrix to display a 2D grid of colors
by making it a list we are providing a 1xNx3 matrix
this displays only one row of colors
N here is the number of clusters

```
print(colors) > creates 2x3 matrix
plt.imshow([colors]) > creates a 1x2x3 matrix
plt.show()
```

output>



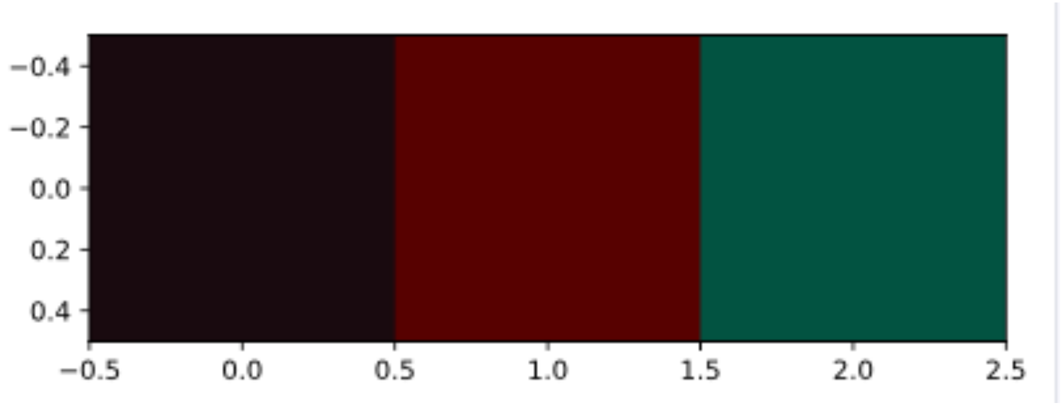
representing this jpeg



Example
```
# 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)
print(batman_image.shape)

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

Example
```
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 DataFrame 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()

# 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()
output>
```

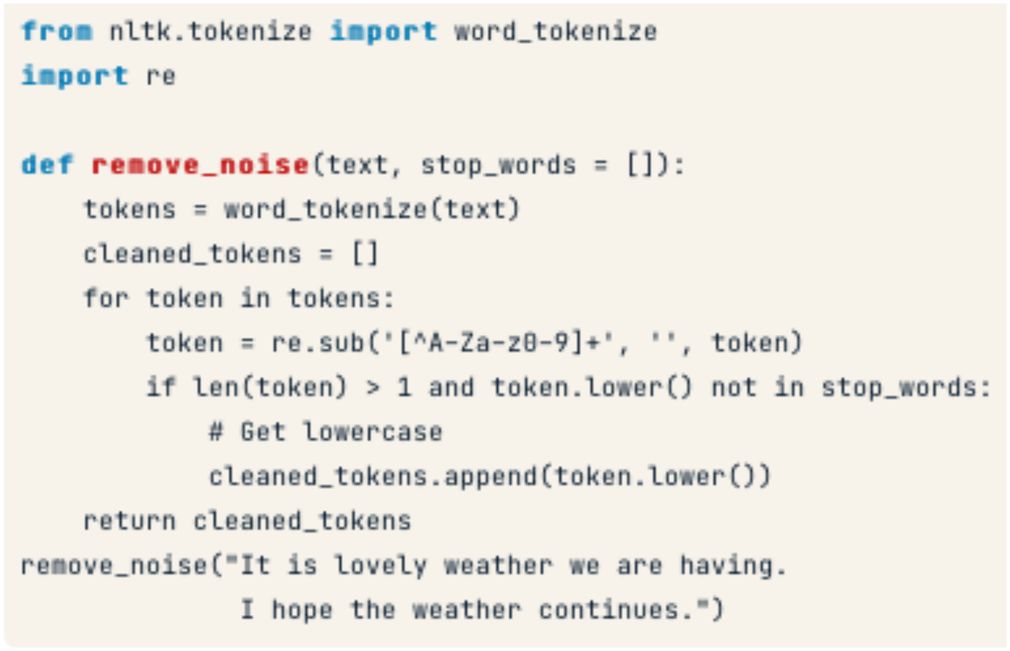


Document clustering
TF-IDF > weighted statistic that describes the importance of a term in a document
(Term Frequency - Inverse Document Frequency)

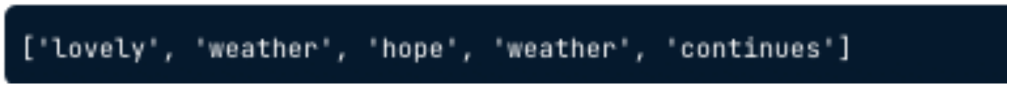
1. clean data (remove anything that will not bring value to the analysis - remove punctuation, ‘the’, etc)

2. determine the importance of the terms in a document (in a TF-IDF matrix)
3. cluster the TF-IDF matrix
4. find top terms and documents in each cluster

Clean and tokenize data
convert text into smaller parts called tokens
clean data for processing
remove all special characters
check for any stop words

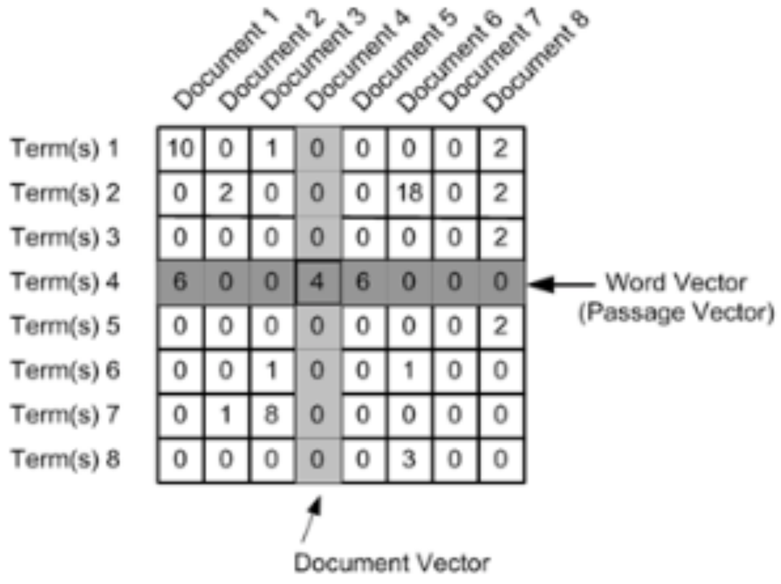


```python
from nltk.tokenize import word_tokenize
import re

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
remove_noise("It is lovely weather we are having.
              I hope the weather continues.")
```



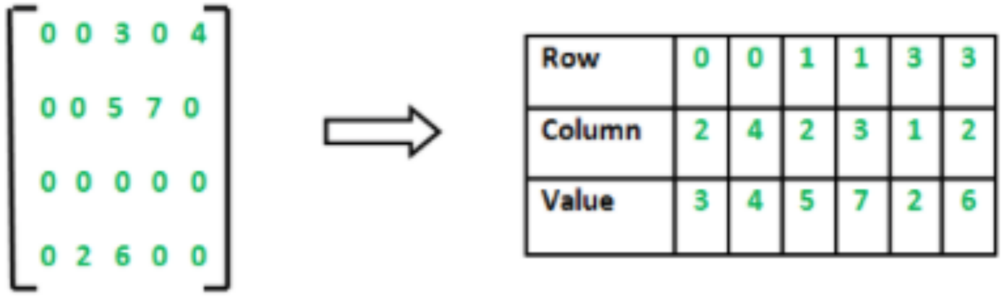
```
['lovely', 'weather', 'hope', 'weather', 'continues']
```

once relevant terms have been extracted, a matrix is formed



	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

Most elements are zeros so we use sparse matrices to store these matrices more efficiently



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

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

*a sparse matrix only contains terms which have non-zero elements

from sklearn.feature_extraction.text import TfidfVectorizer
max_df and min_df signify the max and min fraction of documents a word should occur in
max_features says how many top terms to keep
remove_noise is our custom function from above and we are using that as our tokenizer
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 matrices
we convert the tfidf matrix to its expanded form using the todense() method
cluster_centers, distorition = kmeans(tfidf_matrix.todense(), num_clusters)
*we do not use an elbow plot here because the high number of variables will cause it to take an erratic form

Each cluster center is a list of tfidf weights
which signifies the importance of each term in the matrix
Finding the top terms
create a list of all terms
then create a dictionary where the terms are keys and the tfidf is the values
sort the dictionary by its values in descending order
use the zip method to join two lists in Python
example - 1000 hotel reviews

```
terms = tfidf_vectorizer.get_feature_names_out()
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])
```

output > top three terms from two clusters

Example

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

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

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

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_out()

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])
```

Clustering with multiple features
first create elbow plot, perform clustering, and generate cluster labels

example - using FIFA dataset

```
#cluster centers
print(fifa.groupby(‘cluster_labels’)[[’scaled _heading_accuracy’, ’scaled_volleys’,
‘scaled_finishing’]].mean())
```

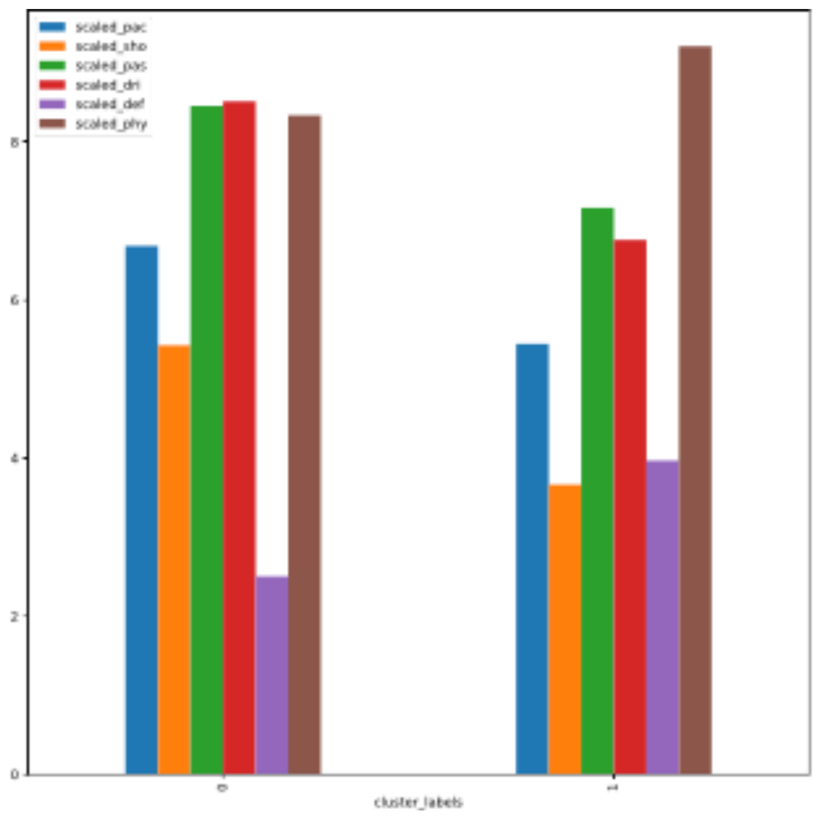
if cluster centers of some features do not vary significantly with respect to the overall data, you may be able to drop that feature in the next run

```
#cluster sizes
print(fifa.groupby(‘cluster_labels’)[‘ID’].count())
```

if one is significantly smaller, you may want to reduce the number of clusters

```
#plot cluster centers
fifa.groupby(‘cluster_labels’) / [scaled_features].mean().plot(kind=‘bar’)
plt.show()

# 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])
```

output>


scaled_pac
scaled_sho
scaled_pas
scaled_dri
scaled_def
scaled_phy
0
2
4
6
8
0
1
cluster_labels