Cluster Analysis

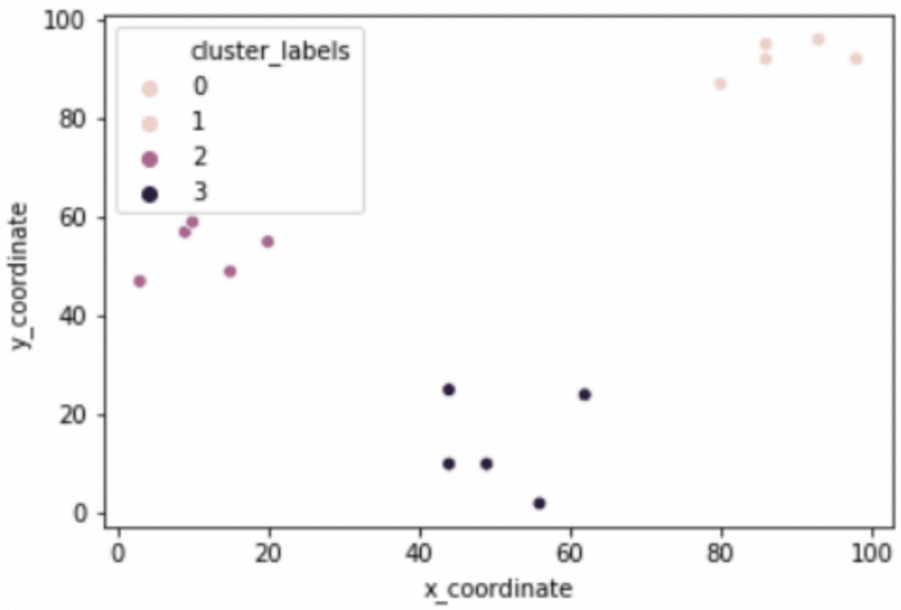
Hierarchical clustering in SciPy

```
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()
```



K-means clustering in SciPy

```
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()
```

Why do we need to prepare data for clustering?

- Variables have incomparable units (product dimensions in cm, price in $)
- Variables with same units have vastly different scales and variances (expenditures on cereals, travel)
- Data in raw form may lead to bias in clustering
- Clusters may be heavily dependent on one variable
- Solution: normalization of individual variables

Normalization of data

Normalization: process of rescaling data to a standard deviation of 1

> x_new = x / std_dev(x)

```
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)
```

Creating a distance matrix using linkage

```
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

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

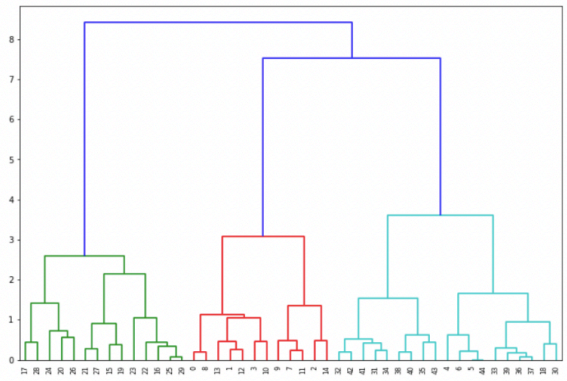
Create cluster labels with fcluster

```
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

Introduction to dendrograms

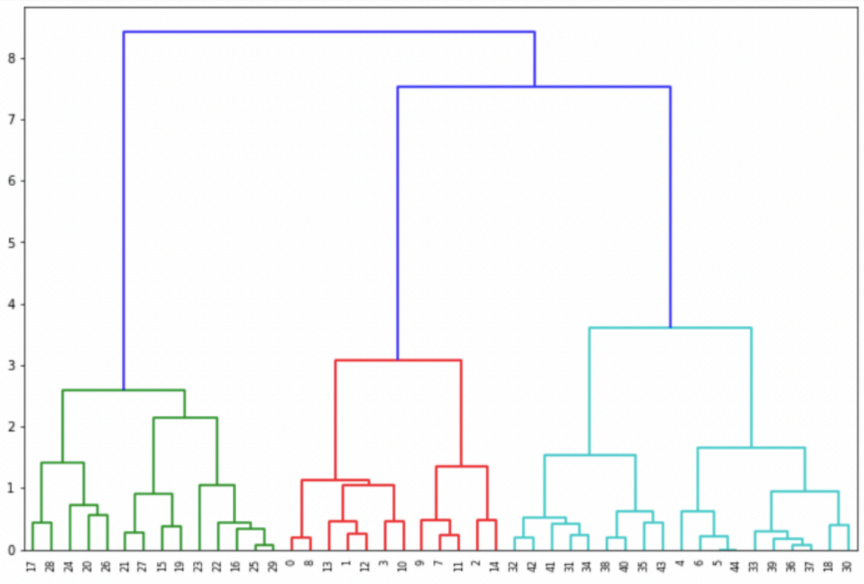
- Strategy till now - decide clusters on visual inspection
- Dendrograms help in showing progressions as clusters are merged
- A dendrogram is a branching diagram that demonstrates how each cluster is composed by branching out into its child nodes



Create a dendrogram in SciPy

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

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



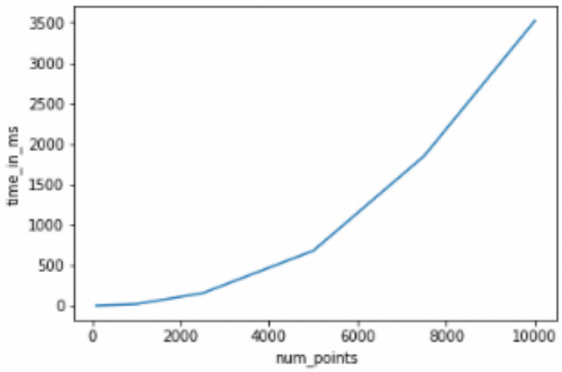
Use of timeit module

```
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)
```

Comparison of runtime of linkage method

- Increasing runtime with data points
- Quadratic increase of runtime
- Not feasible for large datasets



Step 1: Generate cluster centers

```
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 only finite numbers (default: True)

Returns two objects: cluster centers, distortion

Step 2: Generate cluster labels

```
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

Running k-means

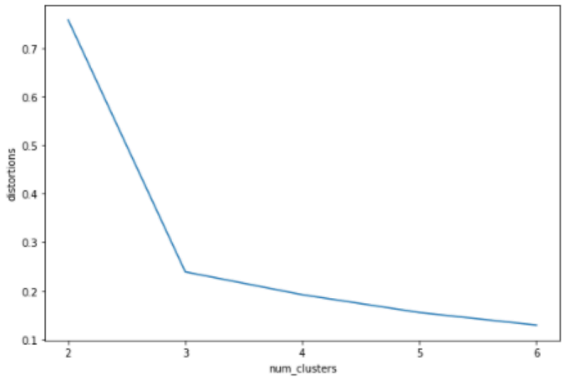
```
# Import kmeans and vq functions
from scipy.cluster.vq import kmeans, vq
```

```
# 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()
```

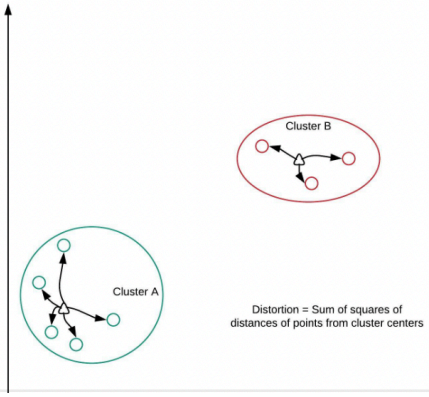
How to find the right k?

- No *absolute* method to find right number of clusters (k) in k-means clustering
- Elbow method



Distortions revisited

- Distortion: sum of squared distances of points from cluster centers
- Decreases with an increasing number of clusters
- Becomes zero when the number of clusters equals the number of points
- Elbow plot: line plot between cluster centers and distortion



Elbow method

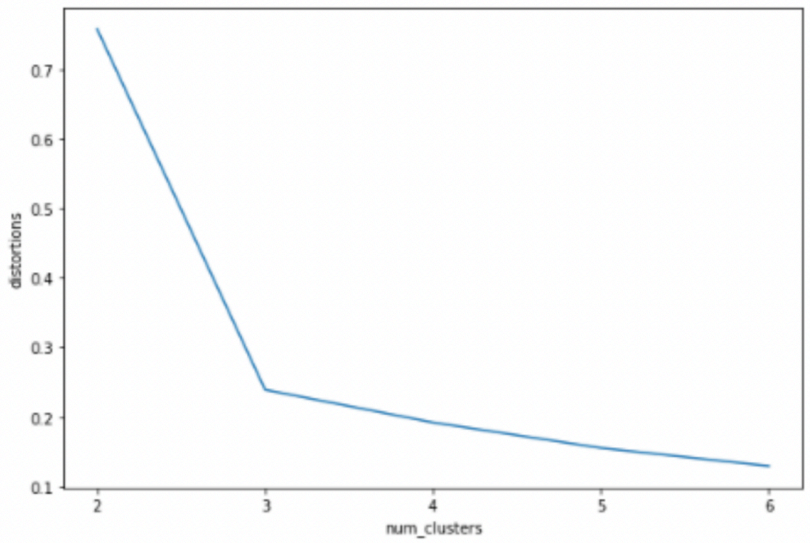
- Elbow plot: plot of the number of clusters and distortion
- Elbow plot helps indicate number of clusters present in data

Elbow method in Python

```
# 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()
```



Final thoughts on using the elbow method

- Only gives an indication of optimal *k* (numbers of clusters)
- Does not always pinpoint how many *k* (numbers of clusters)
- Other methods: average silhouette and gap statistic

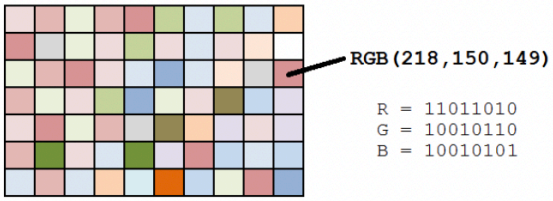
Limitations of k-means clustering

Limitations of k-means clustering

- How to find the right _K_ (number of clusters)?
- Impact of seeds
- Biased towards equal sized clusters

Dominant colors in images

- All images consist of pixels
- Each pixel has three values: *Red*, *Green* and *Blue*
- Pixel color: combination of these RGB values
- Perform k-means on standardized RGB values to find cluster centers
- Uses: Identifying features in satellite images



Source

Tools to find dominant colors

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

Convert image to RGB matrix

```
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)
```

DataFrame with RGB values

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

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

Find dominant colors

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

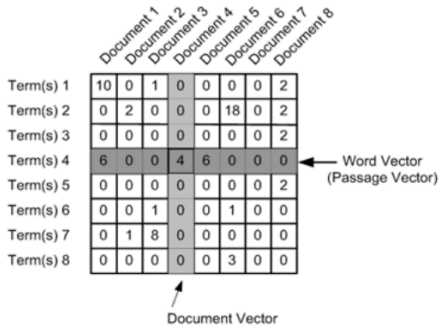
```
colors = []

# 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
    ))
```

Document clustering: concepts

1. Clean data before processing
2. Determine the importance of the terms in a document (in TF-IDF matrix)
3. Cluster the TF-IDF matrix
4. Find top terms, documents in each cluster

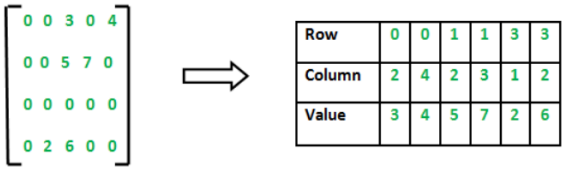
Document term matrix and sparse matrices

- Document term matrix formed
- Most elements in matrix are zeros



Source

- Sparse matrix is created



Source

TF-IDF (Term Frequency - Inverse Document Frequency)

- A weighted measure: evaluate how important a word is to a document in a collection

```
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)
```

Clustering with sparse matrix

- `kmeans()` in SciPy does not support sparse matrices
- Use `.todense()` to convert to a matrix

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

Top terms per cluster

- Cluster centers: lists with a size equal to the number of terms
- Each value in the cluster center is its importance
- Create a dictionary and print top terms

```
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])
```

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

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

More considerations

- Work with hyperlinks, emoticons etc.
- Normalize words (run, ran, running -> run)
- `.todense()` may not work with large datasets