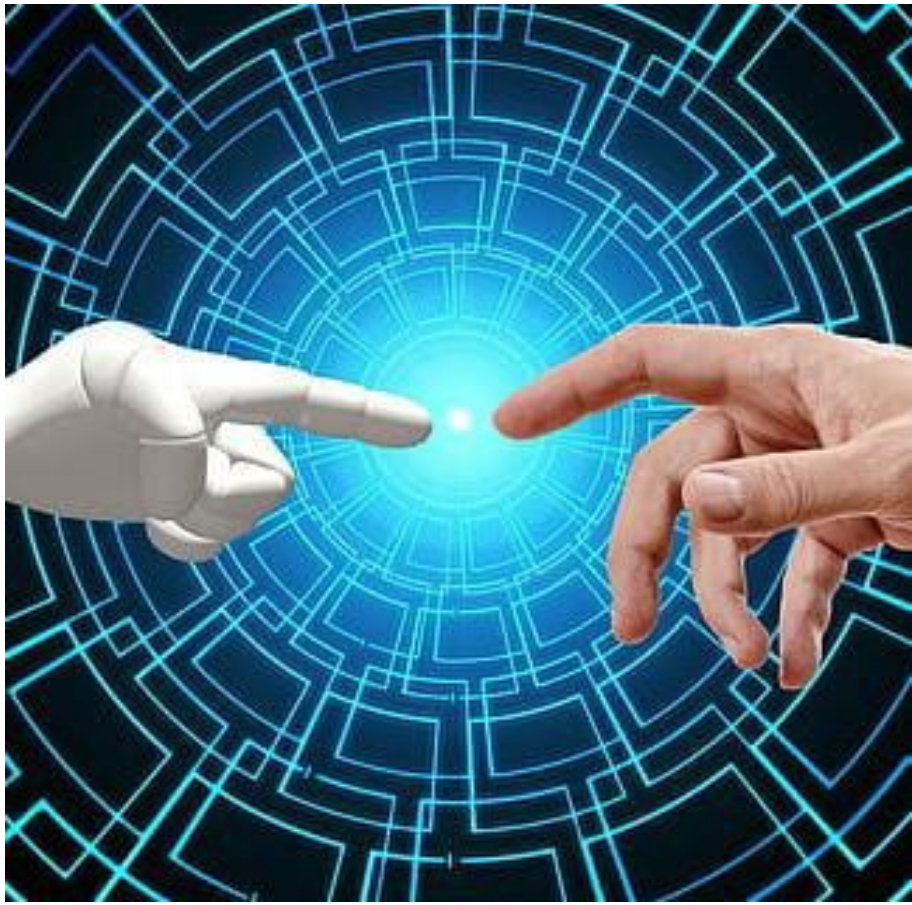
Artificial Intelligence (AI) for Engineering

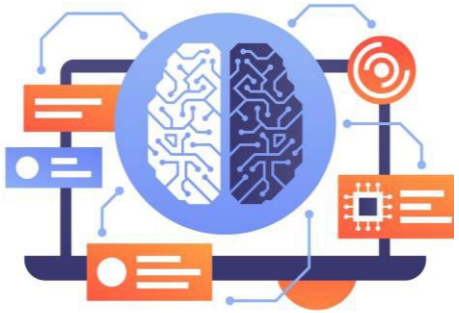


COS40007

Dr. Afzal Azeem Chowdhary

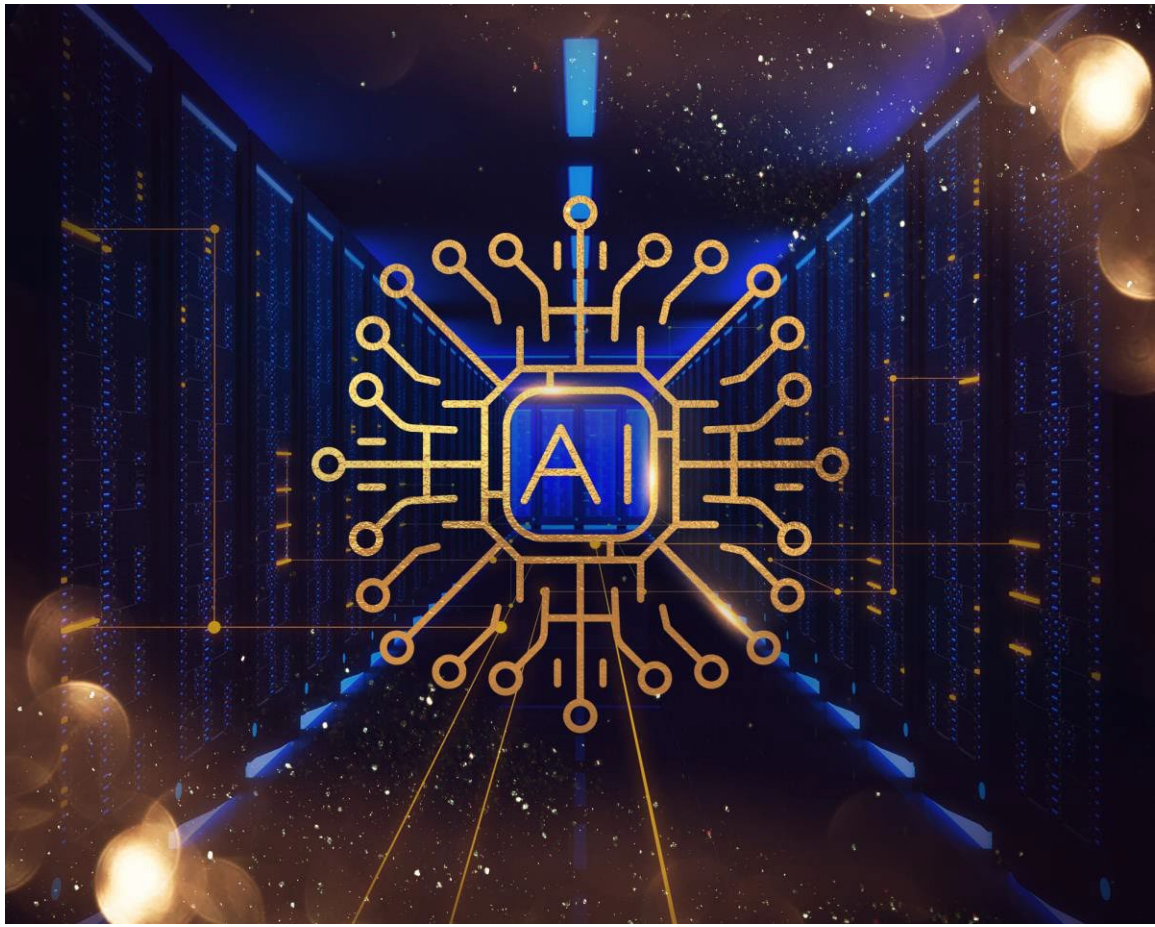
Lecturer, SoCET, Swinburne University of Technology

Seminar 8: 22nd April 2025



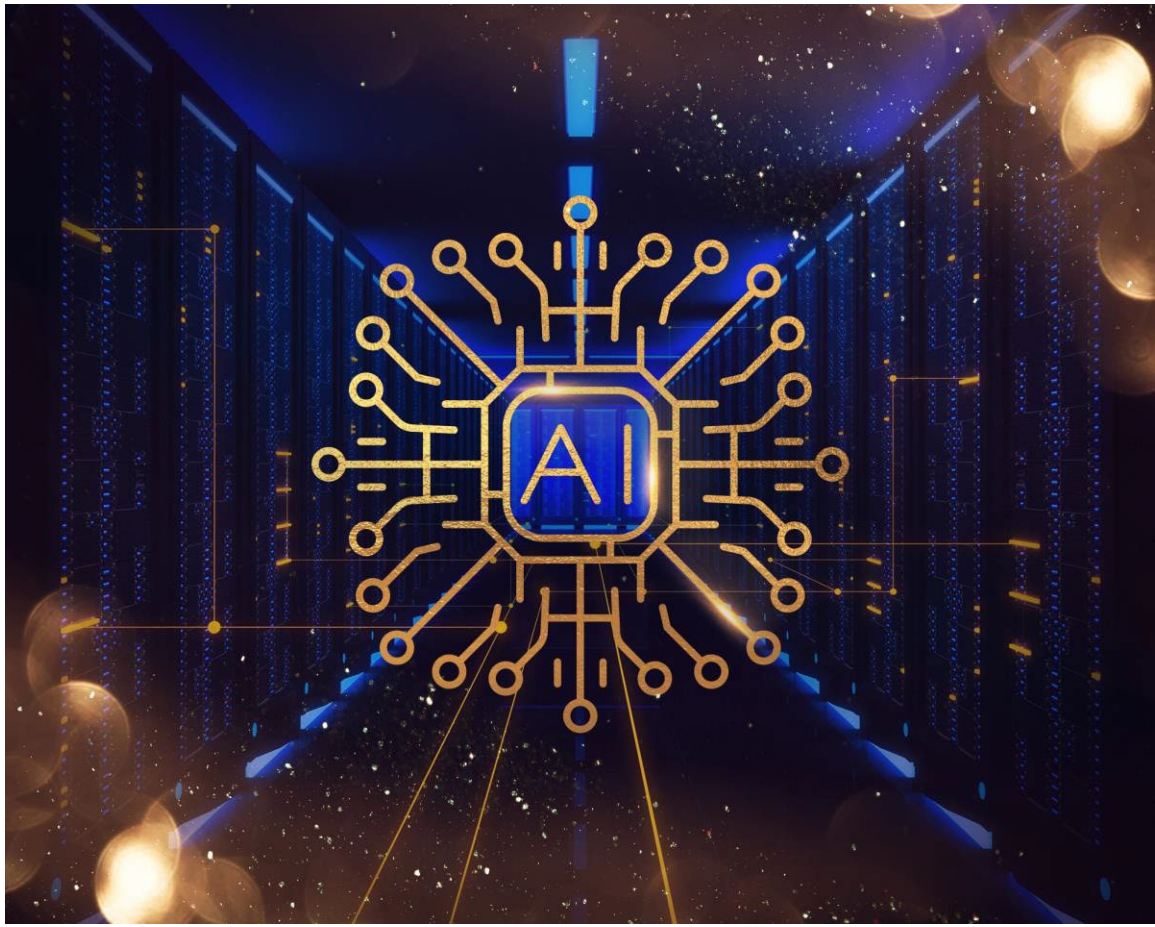
Overview

- ❑ Basics of Clustering
- ❑ Clustering techniques
- ❑ Clustering examples
- ❑ Distance measures for Clustering
- ❑ Deep Learning in Clustering
- ❑ Clustering applications



Required Reading

- Chapter 1 of “Applied Machine Learning and AI for Engineers”
- Chapter 10 of “Machine Learning with PyTorch and Scikit-Learn”



At the end of this you should be able to

- Understand about unsupervised learning
- Understand about data for clustering
- Understand different clustering algorithms
- Understand clustering applications

Clustering

Unsupervised Learning

- Data is not labelled.
- Discover hidden structures in data where we do not know the right answer upfront.
- Finds a natural grouping in data without human intervention.
- Same clusters are more similar to each other than to those from different clusters.
- Clustering is an unsupervised learning approach.

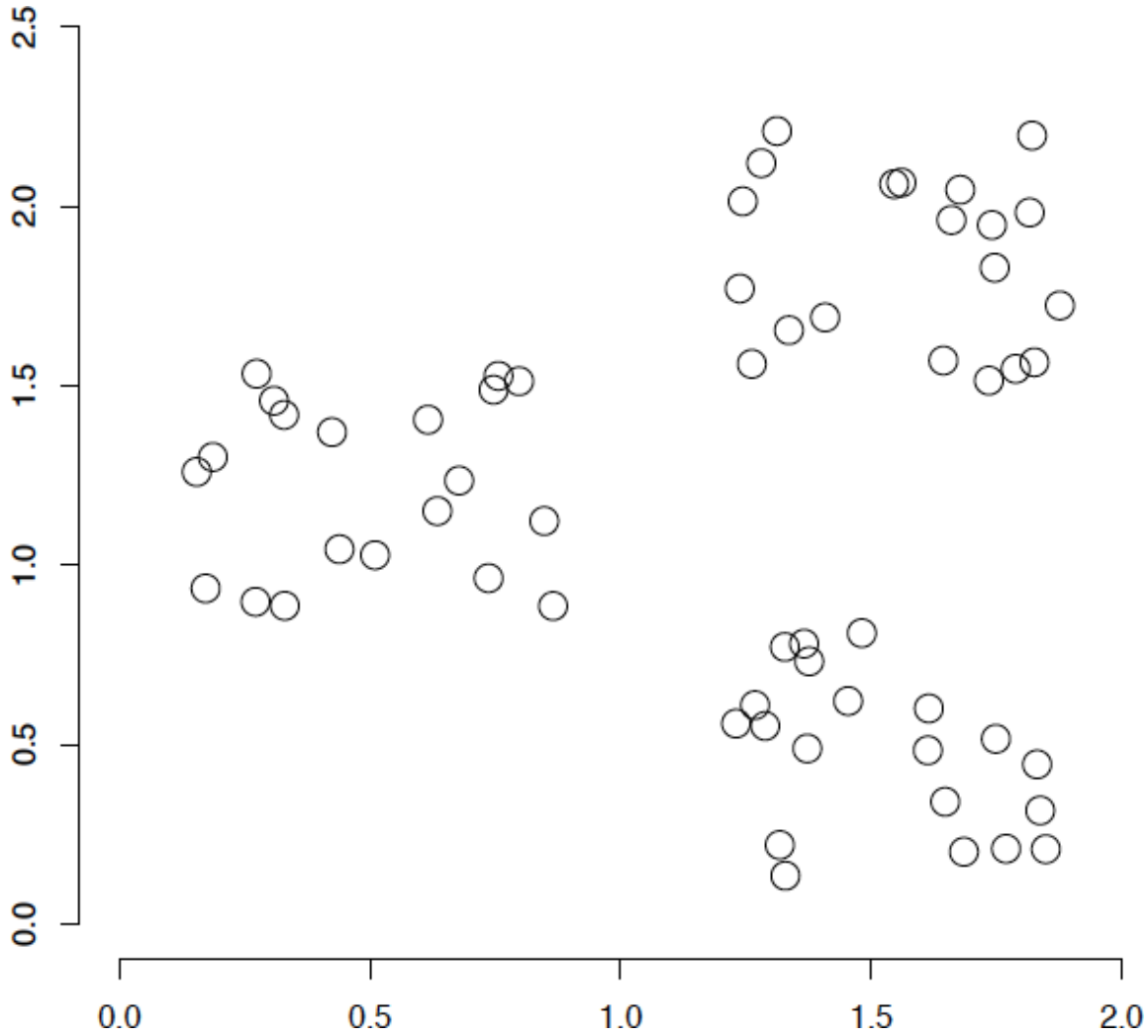
Clustering

Clustering (or cluster analysis) is a technique for finding groups of similar objects that are more related to each other than to objects in different groups.

E.g., finding customers that share similar interests based on common purchase behaviours.

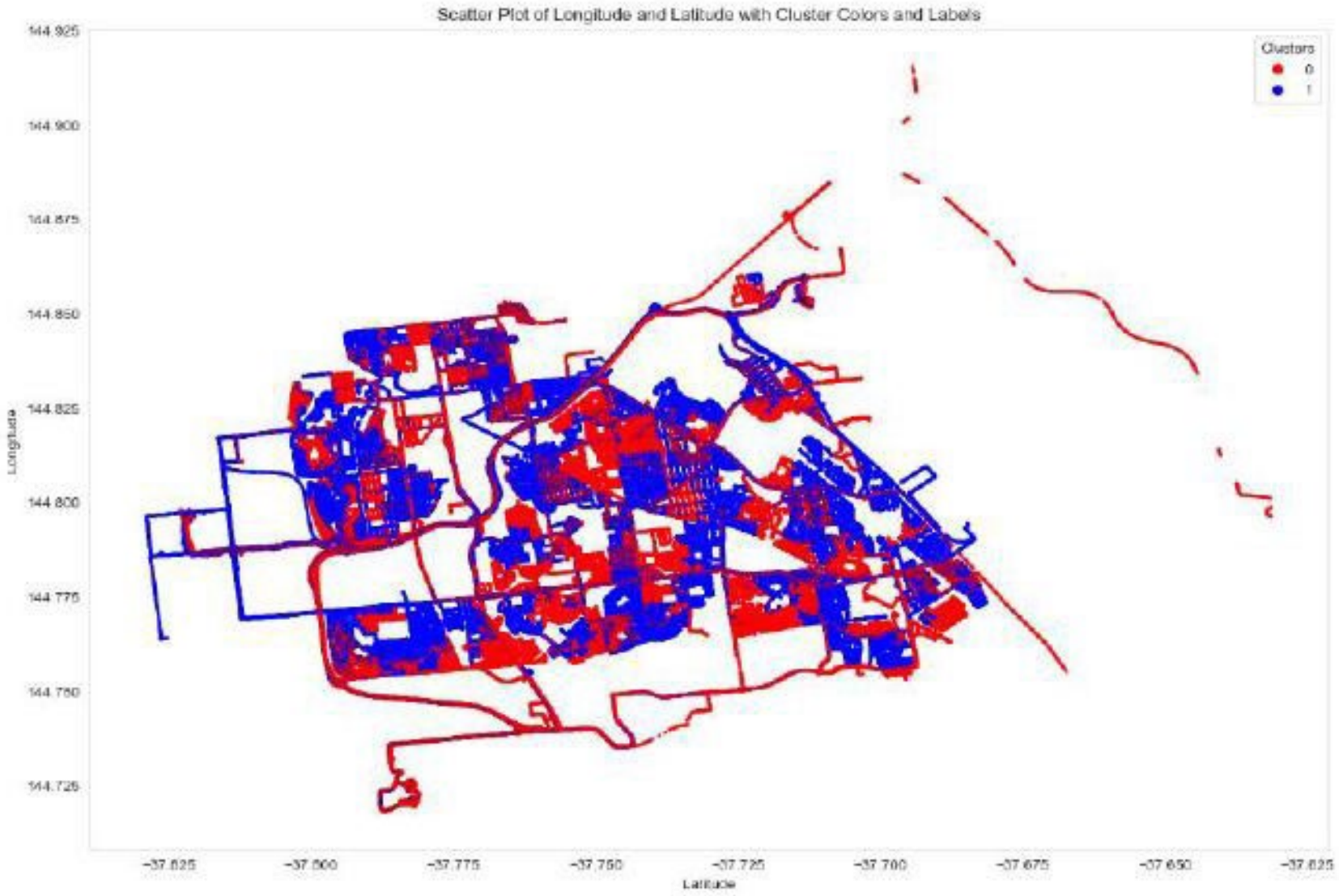
Historical and quality data from a die-casting machine in a production environment are grouped into clusters based on the similarity of the data vectors. Members in each cluster are then analysed to determine operation modes or categories (e.g. ‘high probability of defects)

Data Partitioning Technique

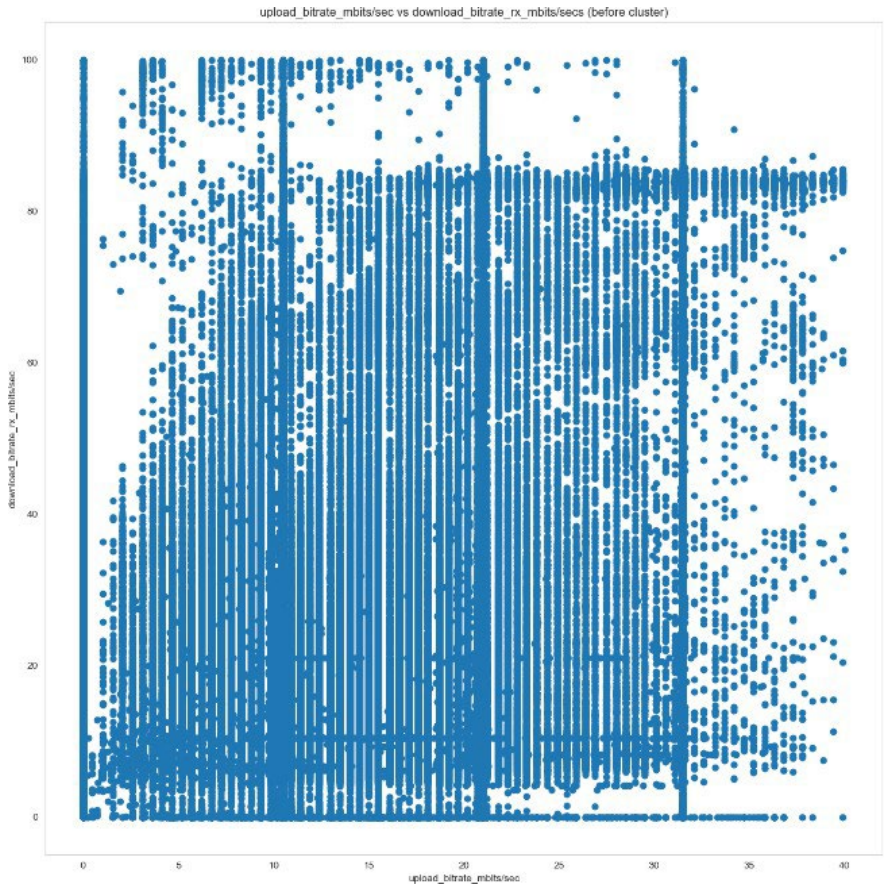


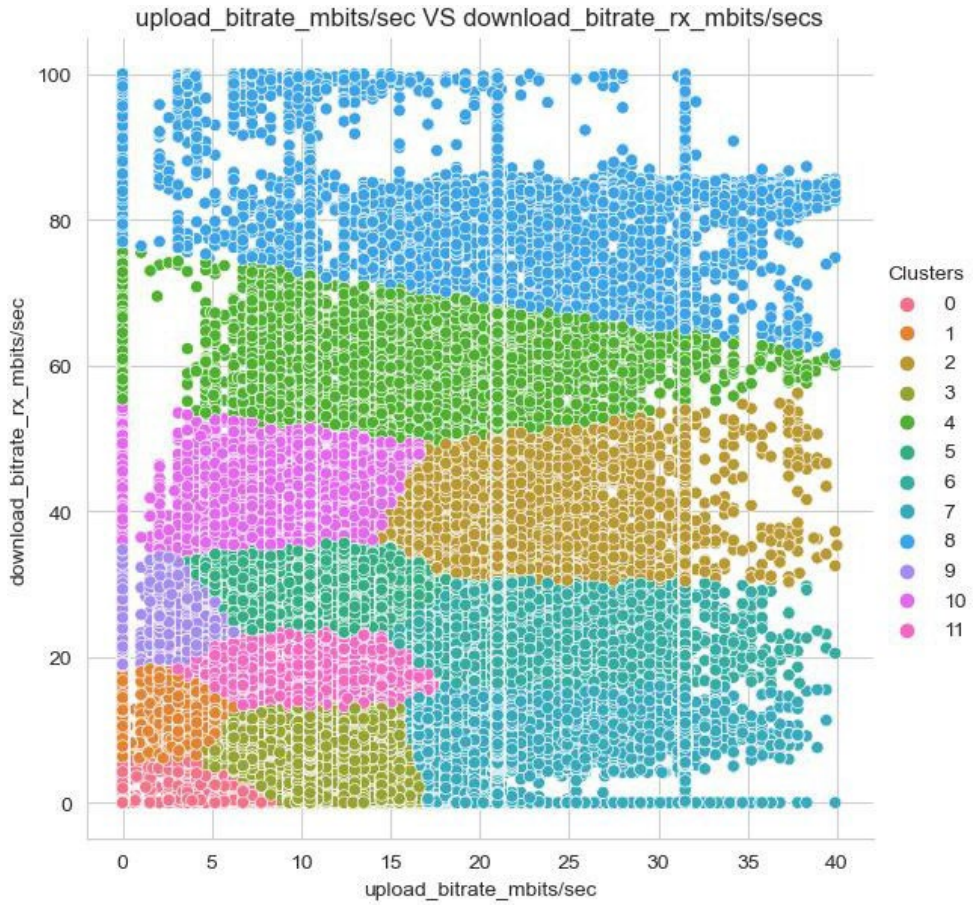
How would you design an algorithm to find the three clusters in this case?

Use of Clustering for Visualization



Clustering


upload_bitrate_mbits/sec vs download_bitrate_rx_mbits/secs (before cluster)
download_bitrate_rx_mbits/sec
upload_bitrate_mbits/sec
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40


upload_bitrate_mbits/sec VS download_bitrate_rx_mbits/secs
download_bitrate_rx_mbits/sec
upload_bitrate_mbits/sec
Clusters
0
1
2
3
4
5
6
7
8
9
10
11
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40

Clustering Algorithms

Flat Algorithms

- Usually start with a random (partial) partitioning
- Refine it iteratively
  - *K* means clustering
  - (Model-based clustering)

Hierarchical Algorithms

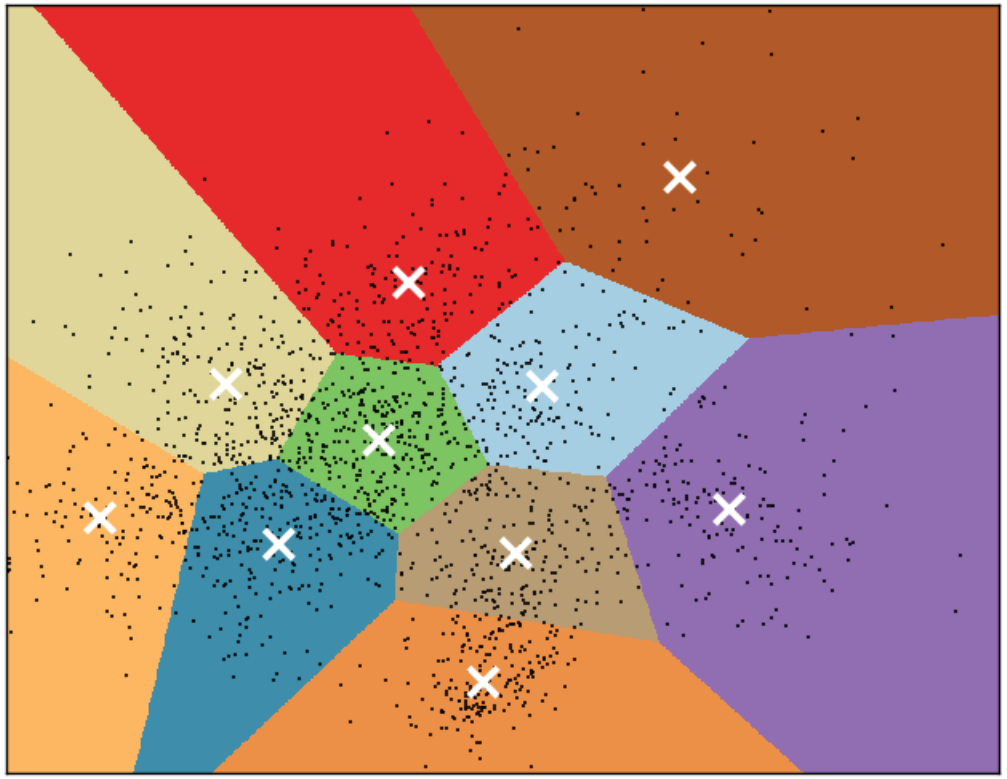
- Bottom-up, Agglomerative
- Top-down, Divisive

Density-based Algorithms

- DBScan

K-Means Clustering

- Formally: a method of vector quantisation
- Partition space into Voronoi cells
- Separate samples into n groups of equal variance
- Uses the Euclidean distance metric



K-Means Clustering

- Assumes documents are real-valued vectors.
- Clusters based on *centroids* (aka the *centre of gravity* or mean) of points in a cluster, *c*:
- Reassignment of instances to clusters is based on the distance to the current cluster centroids.
  - (Or one can equivalently phrase it in terms of similarities)

How Many Clusters?

Number of clusters K is given

- Partition n docs into predetermined number of clusters

Finding the “right” number of clusters is part of the problem

- Given docs/points, partition into an “appropriate” number of subsets.
- E.g., for query results

✓ Ideal value of K not known up front

✓ Though UI may impose limits.

Can usually take an algorithm for one flavor and convert to the other.

K-Means Clustering Steps:

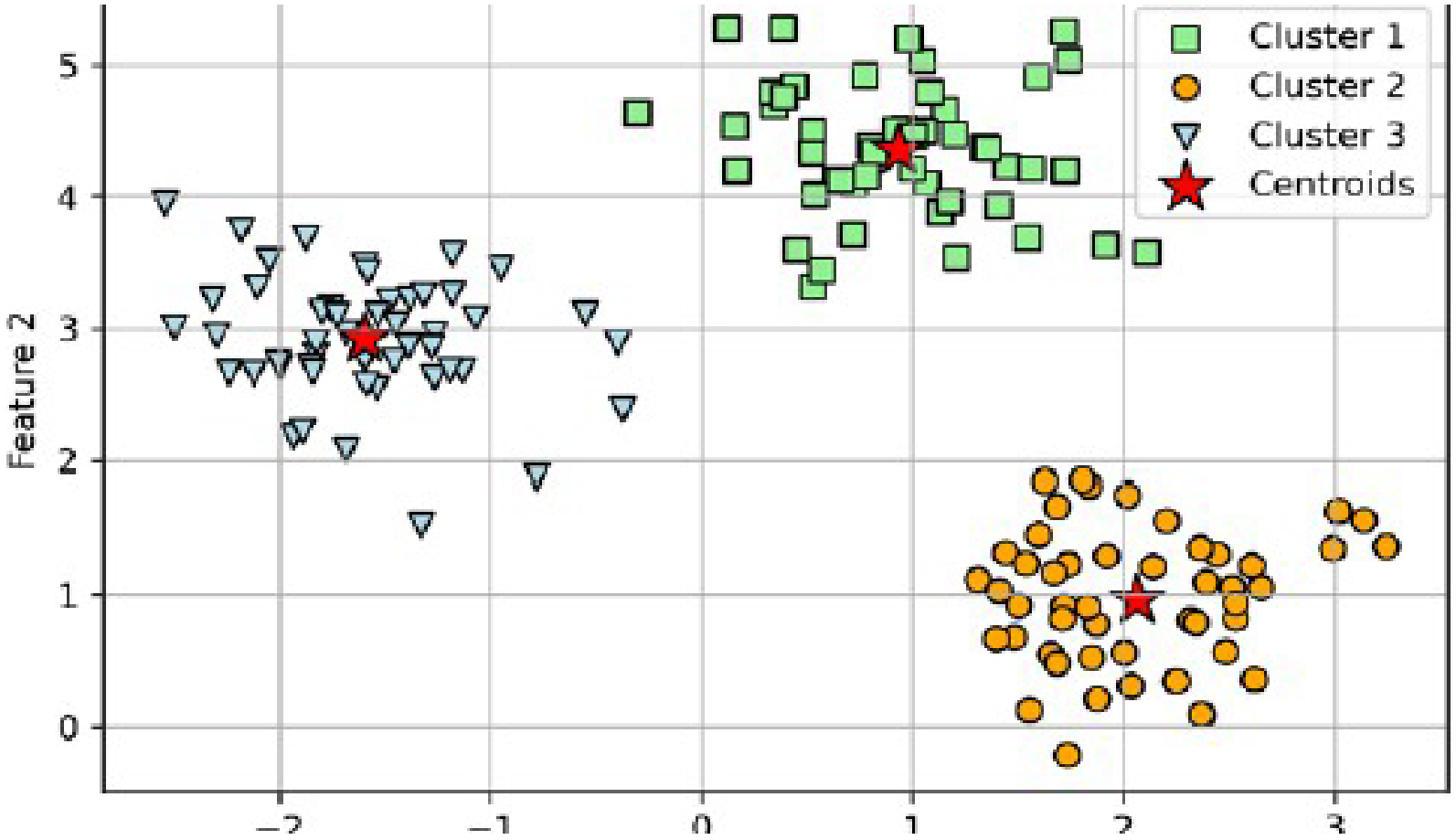
1. Randomly pick k centroids from the examples as initial cluster centers
2. Assign each example to the nearest centroid, $\mu^{(j)}$, $j \in \{1, \ldots, k\}$
3. Move the centroids to the center of the examples that were assigned to it
4. Repeat *steps 2* and *3* until the cluster assignments do not change or a user-defined tolerance or maximum number of iterations is reached

K-means in sklearn

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
... init='random',
... n_init=10,
... max_iter=300,
... tol=1e-04,
... random_state=0)
>>> y_km = km.fit_predict(X)
```

K-means

Cluster can be visulisaed using matplotlib



Basics of *K*-means

Iterative Refinement

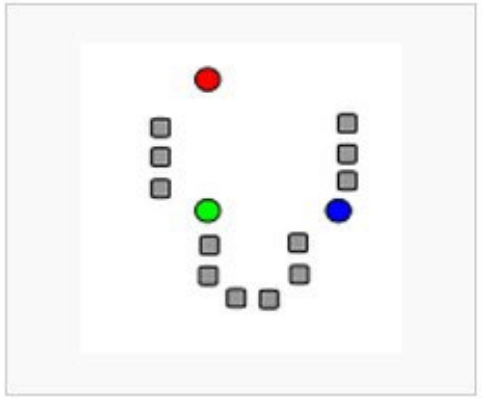
Three basic steps

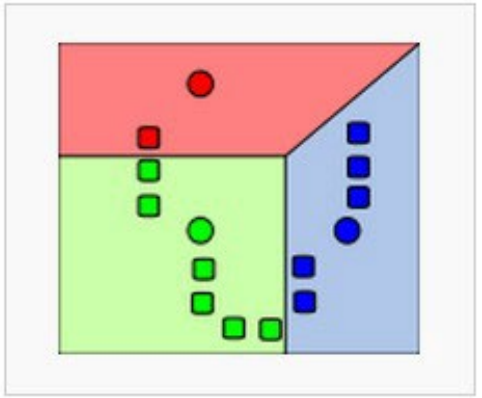
- Step 1: Choose k

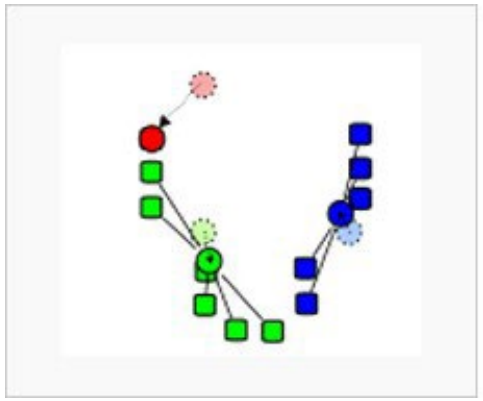
Iterate over

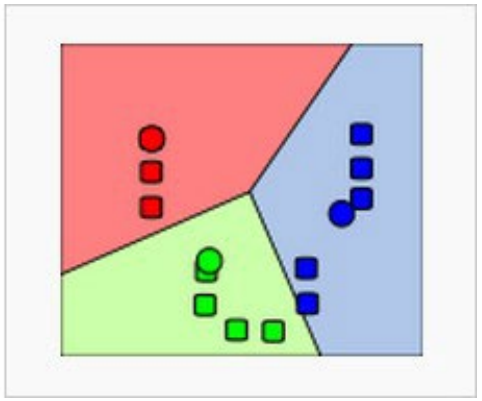
- Step 2: Assignment
- Step 3: Update

Repeats until convergence has been reached







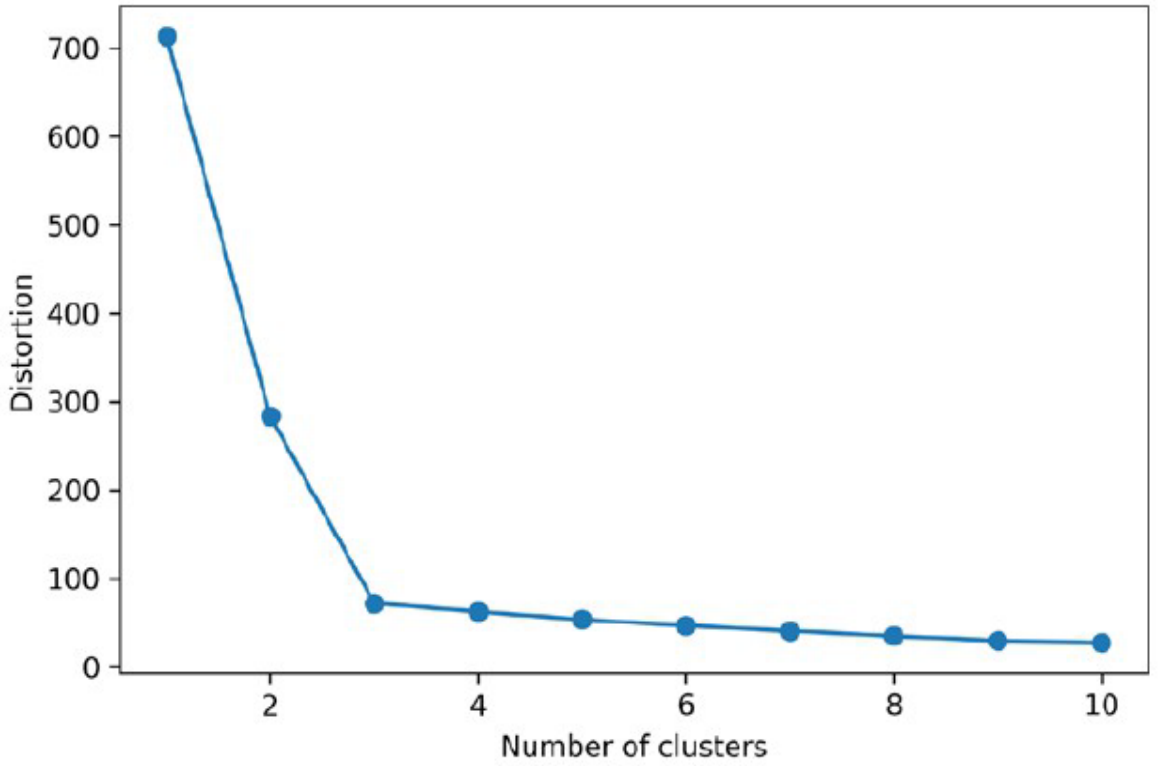


When to use *K*-Means

- Normally distributed data
- Large number of samples
- Not too many clusters
- Distance can be measured in a linear fashion

How to find Optimal Number of Clusters?

- Elbow method is used to find the optimal number of clusters, *k*, for a given dataset
- We need to use intrinsic metrics-such as the within-cluster SSE (distortion) — to compare the performance of different *k*-means clustering models.
- If *k* increases, the distortion will decrease.
- Identify the value of *k* where the distortion begins to increase most rapidly



What is a good clustering?

Internal criterion: A good clustering will produce high-quality clusters in which:

- the intra-class (that is, intra-cluster) similarity is high
- the inter-class similarity is low
- The measured quality of clustering depends on both the document representation and the similarity measure used

Measuring clustering quality: **Silhouette Score**

Silhouette Coefficient

- No ground truth
- Mean distance between an observation and all other points in its cluster
- Mean distance between an observation and all other points in the next nearest cluster

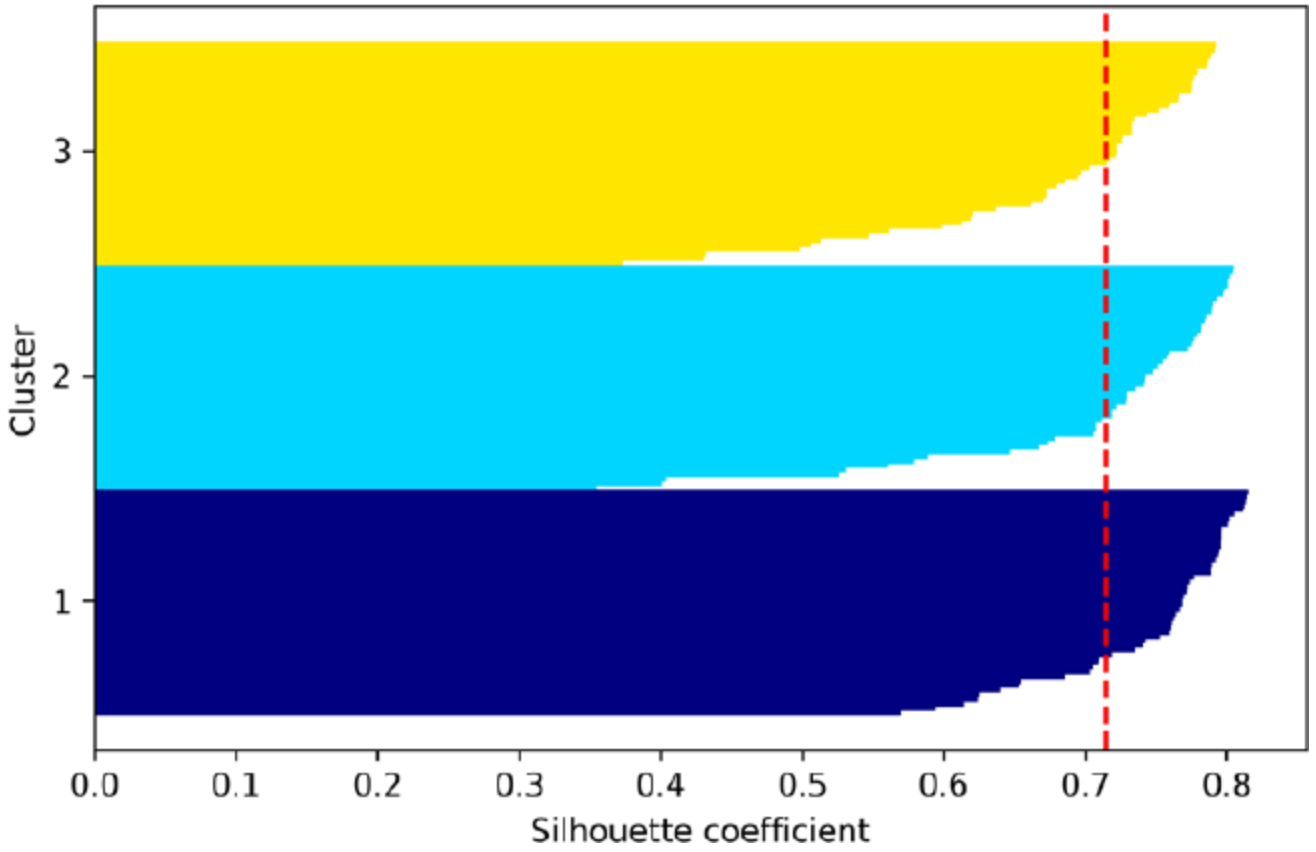
Silhouette score in scikit-learn

- Mean of silhouette coefficient for all of the observations
- Closer to 1, the better the fit
- Large dataset == long time

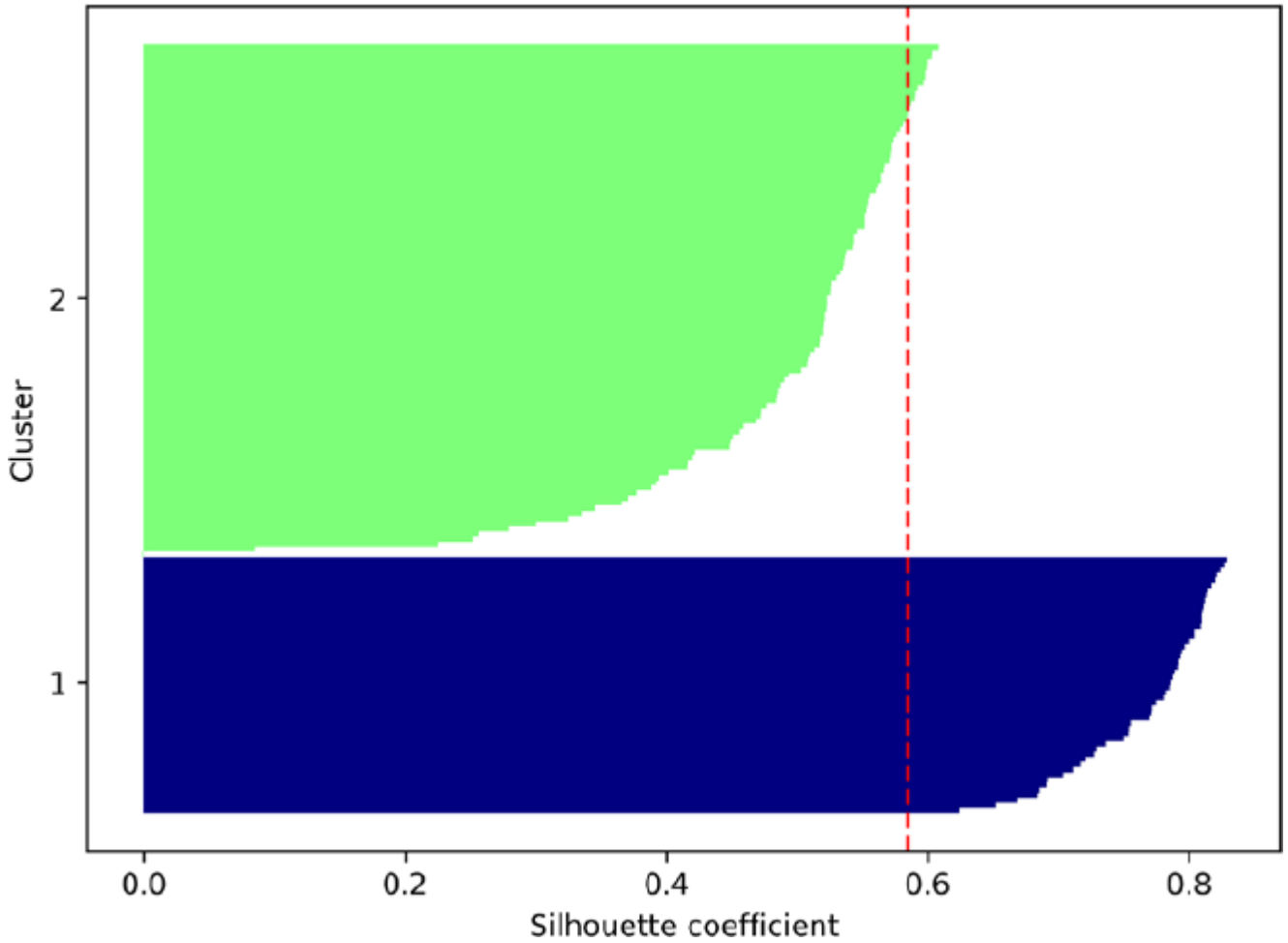
```
>>> km = KMeans(n_clusters=3,
...     init='k-means++',
...     n_init=10,
...     max_iter=300,
...     tol=1e-04,
...     random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(
...  X, y_km, metric='euclidean'
... )
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...  c_silhouette_vals = silhouette_vals[y_km == c]
...  c_silhouette_vals.sort()
...  y_ax_upper += len(c_silhouette_vals)
...  color = cm.jet(float(i) / n_clusters)
...  plt.barh(range(y_ax_lower, y_ax_upper),
...      c_silhouette_vals,
...      height=1.0,
...      edgecolor='none',
...      color=color)
...  yticks.append((y_ax_lower + y_ax_upper) / 2.)
...  y_ax_lower += len(c_silhouette_vals)
```

```
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
... color="red",
... linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

Silhouette score: Example of good clustering

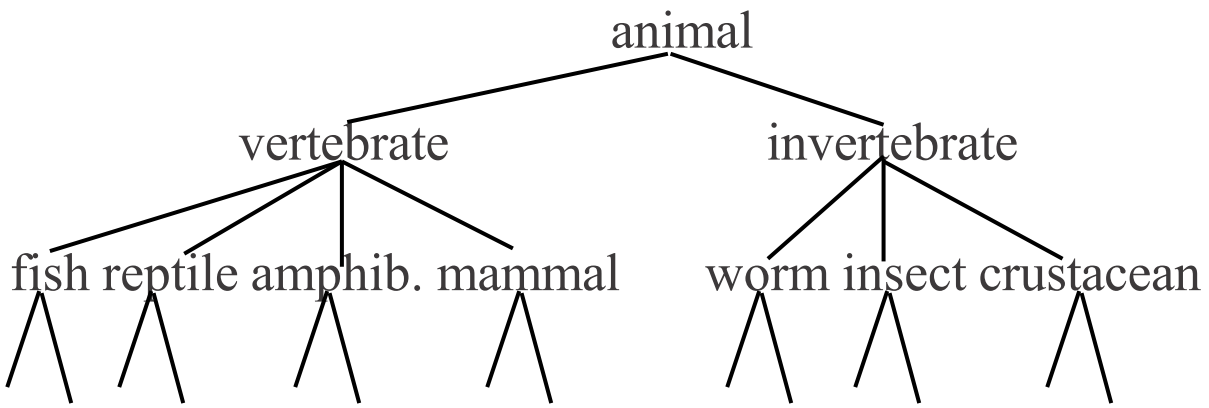


Silhouette score: Example of bad clustering



Hierarchical Clustering

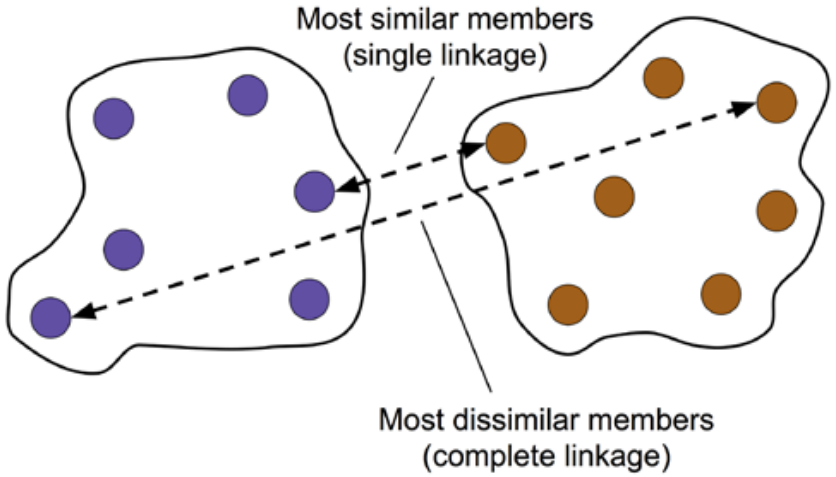
Build a tree-based hierarchical taxonomy (*dendrogram*) from a set of documents.



One approach: recursive application of a partitional clustering algorithm.

Steps of Hierarchical Clustering

1. Compute a pair-wise distance matrix of all examples.
2. Represent each data point as a singleton cluster.
3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
4. Update the cluster linkage matrix.
5. Repeat *steps 2-4* until one single cluster remains.

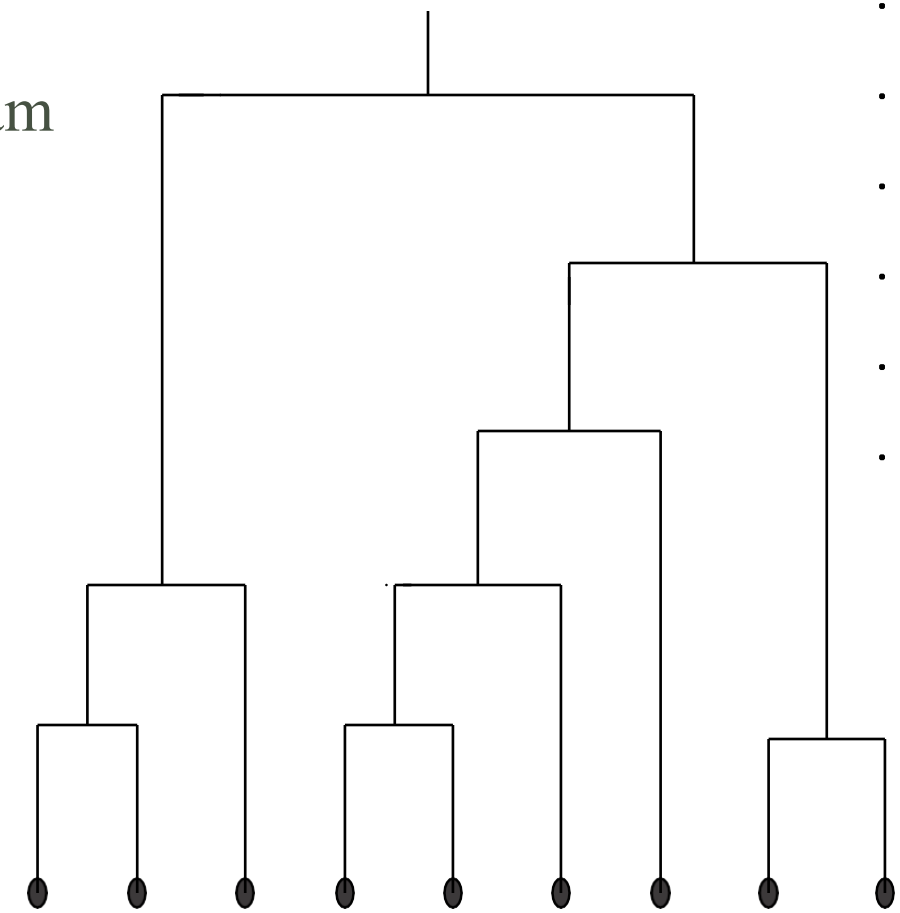


Dendrogram: Hierarchical Clustering

Clustering is obtained by cutting the dendrogram at a desired level: each connected component forms a cluster.

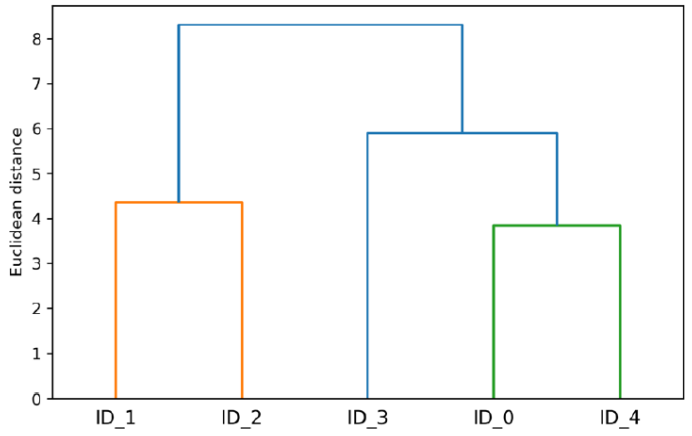
Let's, consider the following example:

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443



Hierarchical Clustering on Distance Metric

	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0.0	4.0	3.835396	2.0
cluster 2	1.0	2.0	4.347073	2.0
cluster 3	3.0	5.0	5.899885	3.0
cluster 4	6.0	7.0	8.316594	5.0



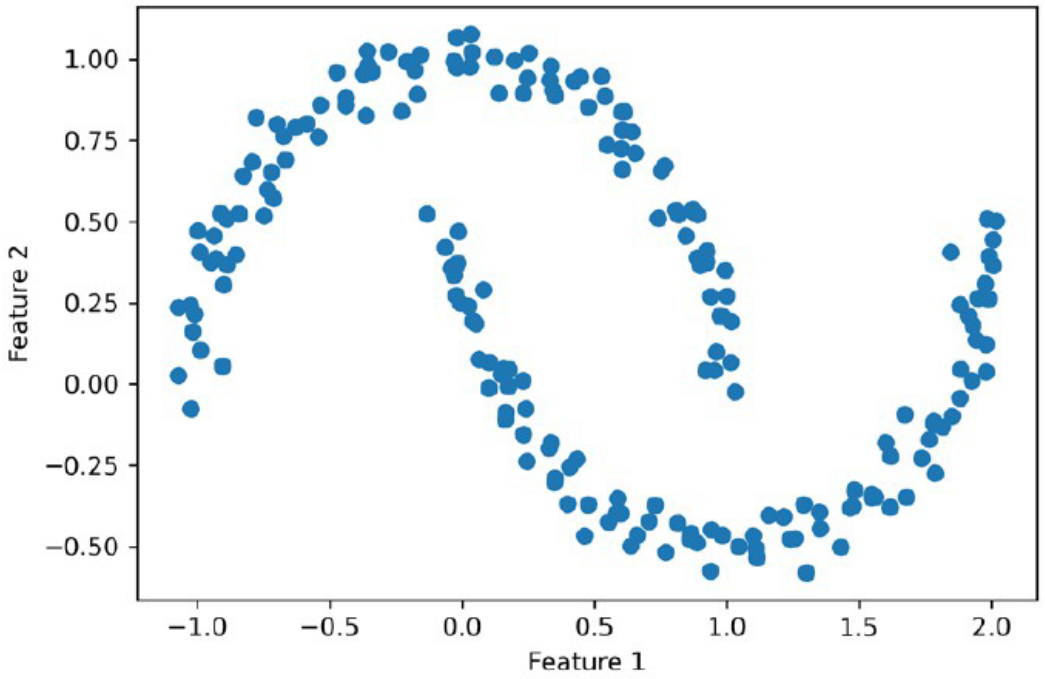
Distance matrix

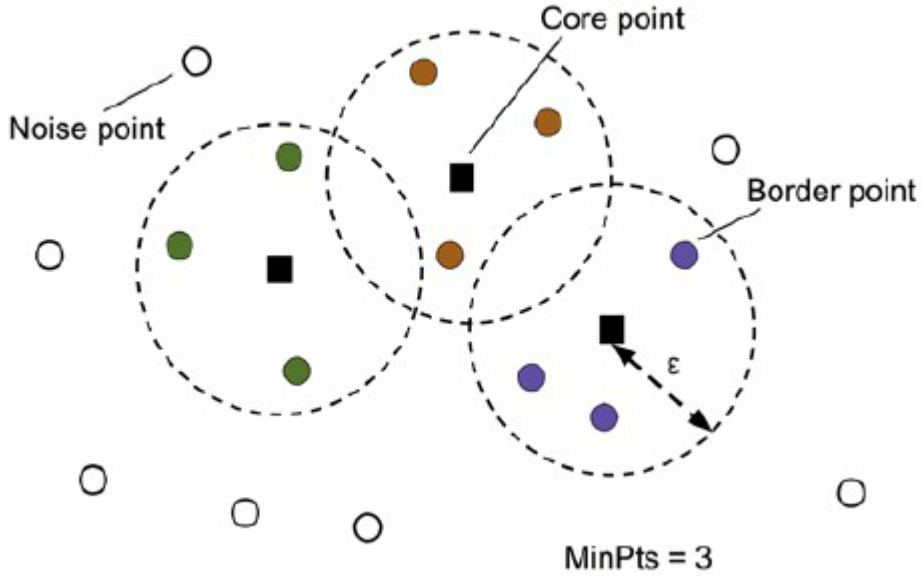
```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...                         pdist(df, metric='euclidean')),
...                         columns=labels, index=labels)
>>> row_dist

>>> from scipy.cluster.hierarchy import dendrogram
>>> # make dendrogram black (part 1/2)
>>> # from scipy.cluster.hierarchy import set_link_color_palette
>>> # set_link_color_palette(['black'])
>>>     row_dendr = dendrogram(
...     row_clusters,
...     labels=labels,
...     # make dendrogram black (part 2/2)
...     # color_threshold=np.inf
... )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Density-based clustering: DBSCAN

- Density-based clustering assigns cluster labels based on dense regions of points.
- One of the main advantages of using DBSCAN is that it does not assume that the clusters have a spherical shape as in k-means.
- Furthermore, DBSCAN is different from *k*-means and hierarchical clustering in that it doesn't necessarily assign each point to a cluster but is capable of removing noise points.





Density-based clustering: DBSCAN Steps

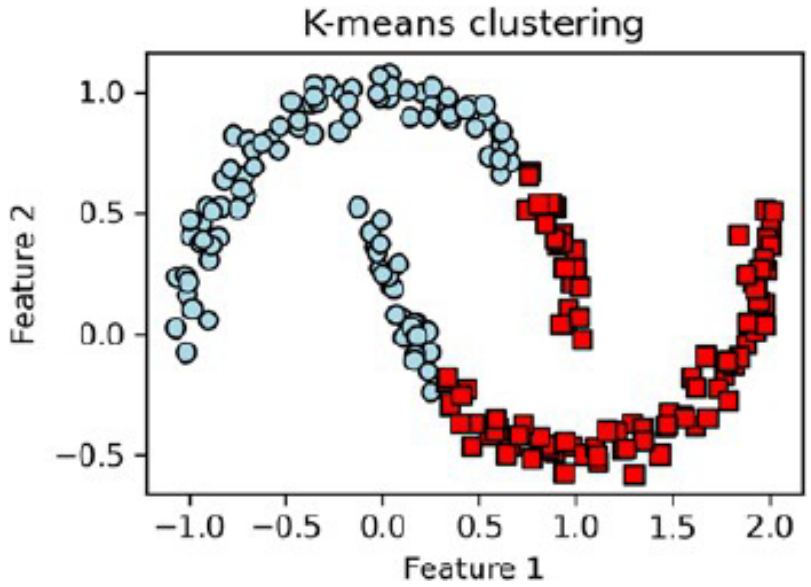
According to the DBSCAN algorithm, a special label is assigned to each example (data point) using the following criteria:

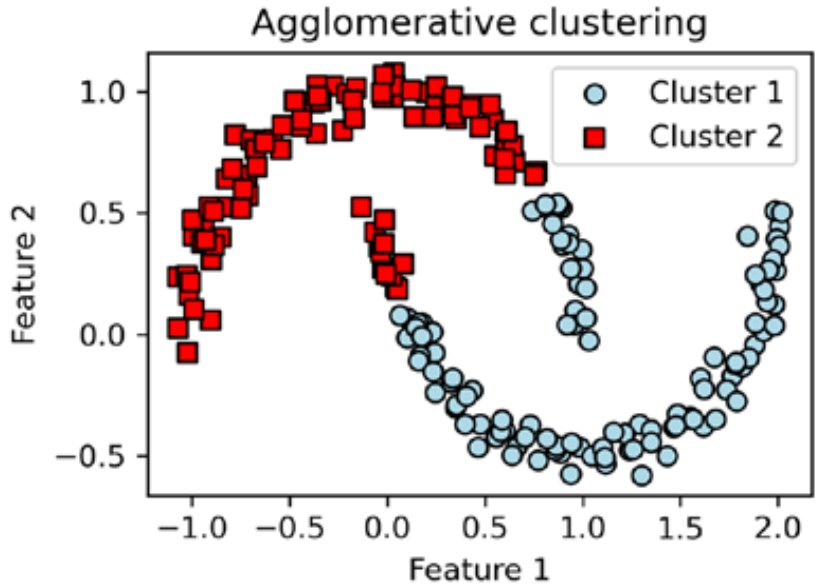
- •A point is considered a core point if at least a specified number (MinPts) of neighboring points fall within the specified radius, ε
- A border point is a point that has fewer neighbors than MinPts within ε , but lies within the ε radius of a core point
- All other points that are neither core nor border points are considered noise points

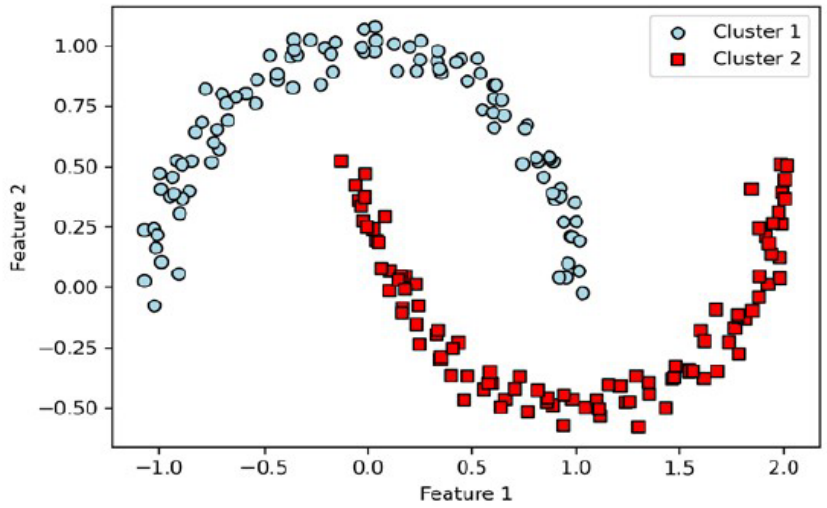
After labeling the points as core, border, or noise, the DBSCAN algorithm can be summarized in two simple steps:

1. Form a separate cluster for each core point or connected group of core points. (Core points are connected if they are no farther away than $\varepsilon\varepsilon$.)
2. Assign each border point to the cluster of its corresponding core point.

DBSCAN







SWINBURNE UNIVERSITY OF TECHNOLOGY

Distance Measures for clustering

- ***Euclidean Distance:*** most commonly used, calculates the straight-line distance between two points in n-dimensional space.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- ***Manhattan Distance:*** the total of the absolute differences between their Cartesian coordinates.

$$d = |x_2 - x_1| + |y_2 - y_1|$$

- ***Cosine Similarity:*** calculates the cosine of the angle between two data points, with a higher cosine value indicating greater similarity.

$$\cos(\vartheta) = A \cdot B / \| A \| \| B \|$$

- ***Minkowski Distance:*** a generalised form of both Euclidean and Manhattan distances. $d = (_{i\ =\ 1}\sum^{n}|x_i - y_i|^p)^{1/p}$

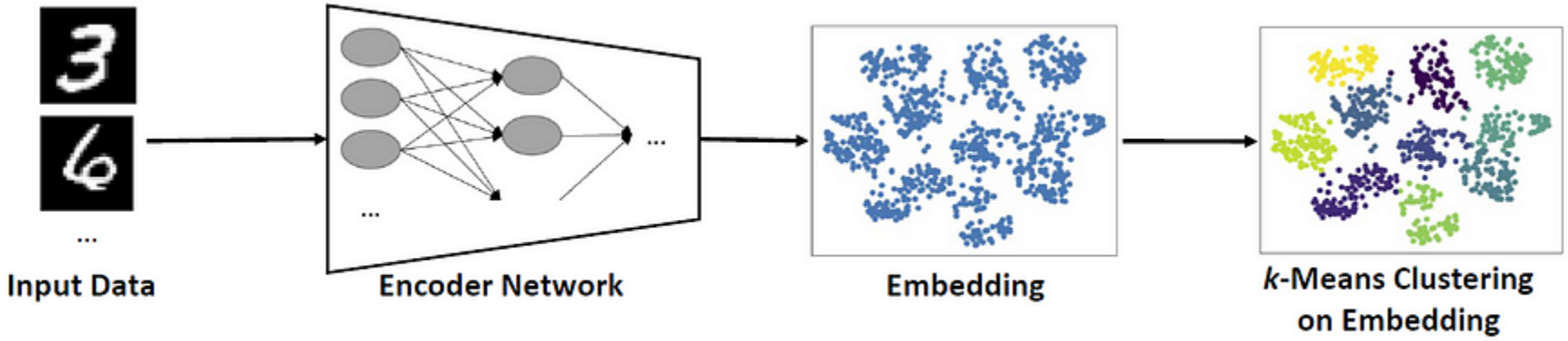
- ***Jaccard Index:*** calculates the ratio of the features shared by two data points to the total number of features.

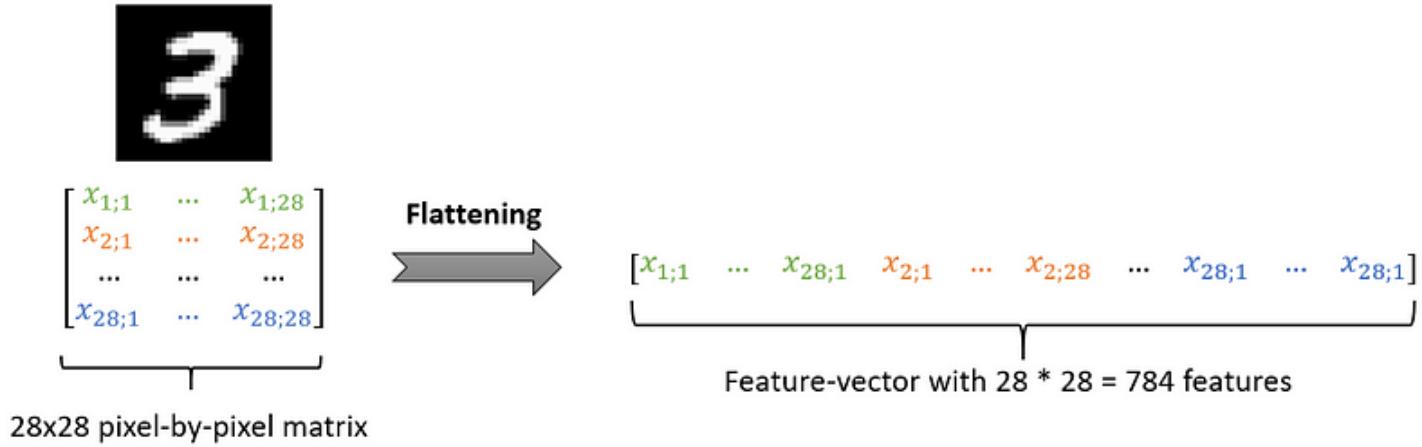
$$J(A,B) = | A \cap B | / | A \cup B |$$

SWINBURNE UNIVERSITY OF TECHNOLOGY

Clustering High-Dimensional Data: Auto-Encoder

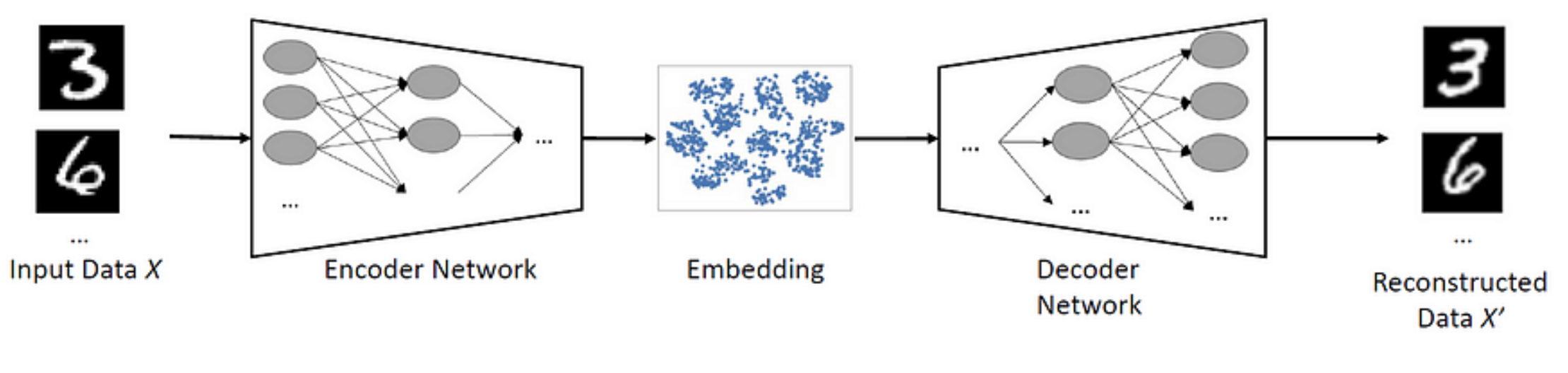
- Auto-encoders can effectively reduce the dimensionality of the data to improve the accuracy of the subsequent clustering
- Example: Automatically group similar images in the same clusters





https://medium.com/@tschechd/deep-auto-encoders-for-clustering-understanding-and-implementing-in-pytorch-8cc748a5fa48

Auto-Encoder




```
n_input_features = 784  # Dimension of MNIST dataset

# Define the AutoEncoder model
model = AutoEncoder(
    input_size=n_input_features,
    hidden_layers=[500, 500, 2000, 10],  # Corrected list of neurons in hidden layers
    dropout_rate=0.2  # Prevent overfitting by deactivating 20% of the neurons
)
```

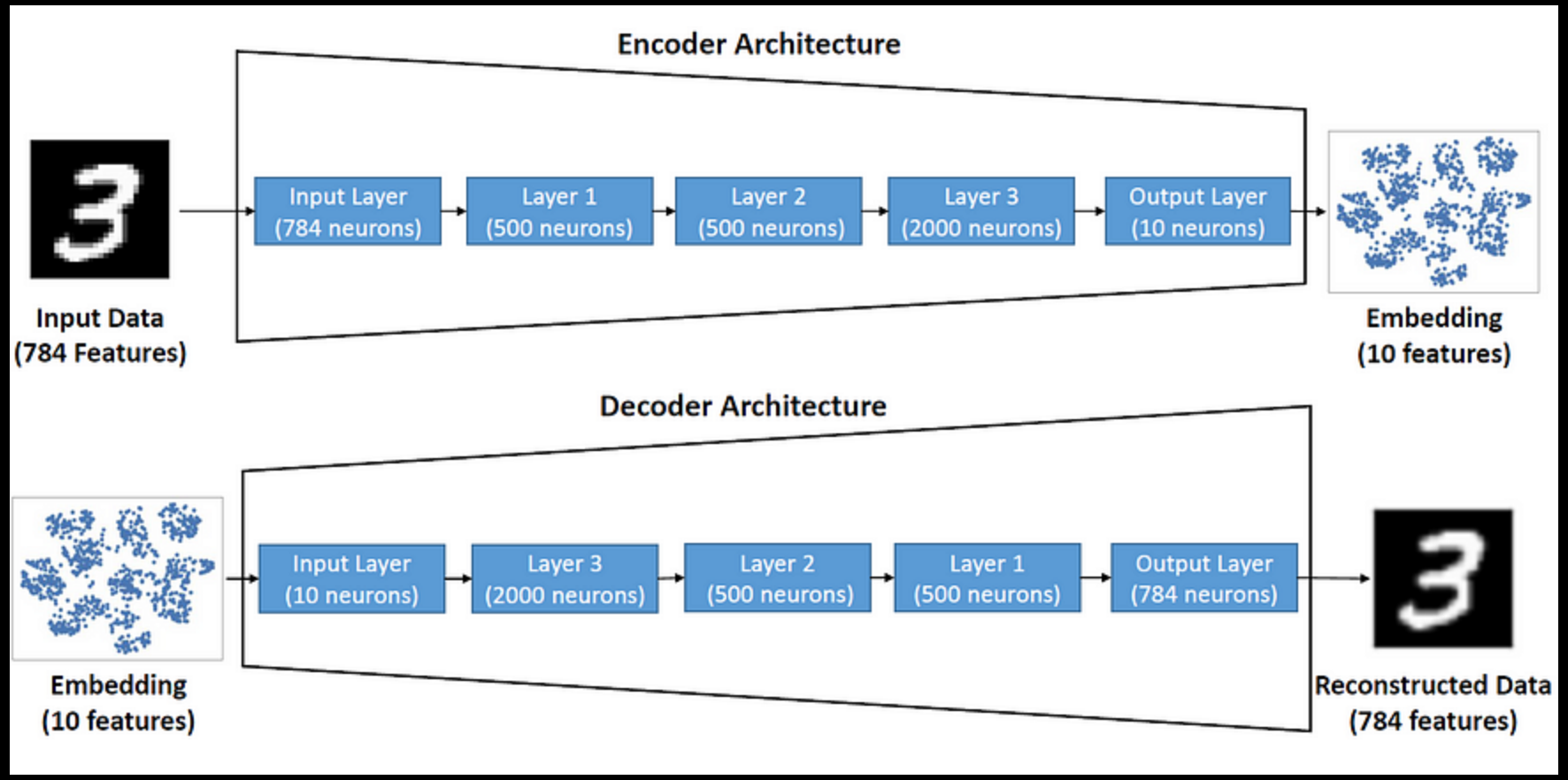
Auto-Encoder Architecture

1. First, we have to load the data.
2. Second, we pre-train the model, i.e., this is a normal training procedure.
3. Finally, we use fine-tuning to improve the performance of our model, which is also a training procedure with a slightly different parameter setting.

During each training epoch, we perform the following four steps to train our model:

a. Apply our model to get the result of the decoder, i.e., the reconstructed data X'.
b. Calculate the reconstruction loss $|X — X'|^2$
c. Backpropagate the loss through the decoder and then through the encoder network, i.e., calculate the respective gradients
d. Update the weights based on the calculated gradients

Auto-Encoder Architecture



Autoencoder

```
from torchvision.datasets import MNIST
from torch.utils.data import ConcatDataset
from torchvision import transforms
# Transform into a tensor and apply normalization
transform = transforms.Compose([transforms.ToTensor(),
                transforms.Normalize((0.5,), (0.5,)),
])
# Load Training Data
trainset = MNIST('./', download=True,
                train=True,
                transform=transform)# Load Test data
                testset = MNIST('./', download=True,
                train=False,
                transform=transform)# Combine both training and test data
dataset = ConcatDataset([trainset, testset])# Use a Data loader, this does not
load the whole dataset at once
# but we can traverse it in batches for training
dataloader = torch.utils.data.DataLoader(dataset,
                batch_size=256,
                shuffle=True,
                num_workers=10)
```

```
X_train = trainset.data.numpy().reshape(60000, 784)
X_test = testset.data.numpy().reshape(10000, 784)
y_train = np.array(trainset.targets)
y_test = np.array(testset.targets)
# Concatenate training and test data
y = np.concatenate([y_train, y_test])
X = np.concatenate([X_train, X_test])
# Convert X to Tensor object
X_tensor = Tensor(X).to(device)
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Evaluating Autoencoder

Clustering Approach	AMI	ARI
k-Means	50.0	36.7
Auto-Encoder (pre-trained)	55.2	47.2
Auto-Encoder (fine-tuned)	70.7	63.6

To evaluate clustering-accuracy,

- Adjusted Mutual Information (AMI)
- Adjusted Rand Index (ARI)

Both are used in many works for unsupervised clustering and compare whether pairwise instances belong to the same cluster in the predictions and in the ground-truth labels.

Auto-Encoder

- Auto-Encoders are powerful, unsupervised deep learning networks to learn a lower-dimensional representation
- Use specifically in image data
- Auto-encoder improves the performance of *k*-means

Clustering applications

- Understanding different types of defects/anomalies in production
- Clustering customers/consumers
- Understand similar trends in data
- Grouping locations on map based on features
- Clustering users in a social network
- Clustering articles based on topics
- Clustering photographs by content, grouping medical images based on visual features
- Clustering music tracks, and video footages

Learn, Practice and Enjoy the AI journey