Unsupervised Machine Learning

- **Unsupervised learning** is a type of machine learning that looks for previously undetected patterns in a data set with no pre-existing labels and with a minimum of human supervision.
- Classical Unsupervised algorithms:
  - K - Means Clustering
  - Hierarchical Tree
  - K – Nearest Neighbor

K – Means Clustering

***K*-means clustering** is a method aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

K-means algorithm

The K-means algorithm starts by randomly choosing a centroid value for each cluster. After that the algorithm iteratively performs three steps:

(i) Find the Euclidean distance between each data instance and centroids of all the clusters;

(ii) Assign the data instances to the cluster of the centroid with nearest distance;

(iii) Calculate new centroid values based on the mean values of the coordinates of all the data instances from the corresponding cluster.

K-means clustering example

- Iris dataset: 4 features and 3 species.
- K – means clustering:
  - Finds clusters of samples
  - Number of clusters must be specified
  - Implemented in sklearn

```
print(samples)
```

```
[[ 5.   3.3  1.4  0.2]
 [ 5.   3.5  1.3  0.3]
 ...
 [ 7.2  3.2  6.   1.8]]
```

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3)
model.fit(samples)
```

```
KMeans(algorithm='auto', ...)
```

```
labels = model.predict(samples)
print(labels)
```

```
[0 0 1 1 0 1 2 1 0 1 ...]
```

Cluster labels for new samples

- New samples can be assigned to existing clusters
- k-means remembers the mean of each cluster (the "centroids")
- Finds the nearest centroid to each new sample

```
print(new_samples)
```

```
[[ 5.7  4.4  1.5  0.4]
 [ 6.5  3.   5.5  1.8]
 [ 5.8  2.7  5.1  1.9]]
```

```
new_labels = model.predict(new_samples)
print(new_labels)
```

```
[0 2 1]
```

Cluster evaluation

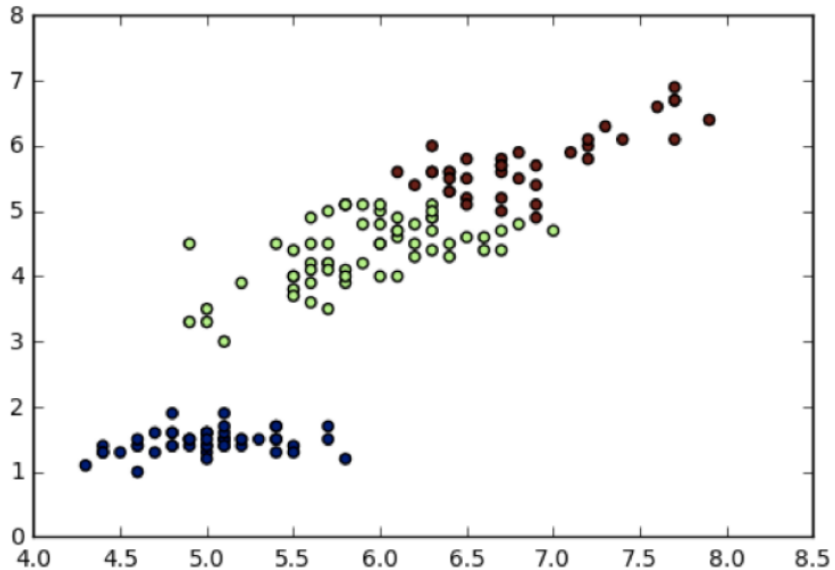
Iris: clusters vs species

- k-means found 3 clusters amongst the iris samples
- Do the clusters correspond to the species?

species	setosa	versicolor	virginica
labels			
0	0	2	36
1	50	0	0
2	0	48	14

Scatter plots

- Scatter plot of sepal length vs. petal length
- Each point represents an iris sample
- Color points by cluster labels
- PyPlot (`matplotlib.pyplot`)



Measuring clustering quality

- Using only samples and their cluster labels
- A good clustering has tight clusters
- Samples in each cluster bunched together

Inertia measures clustering quality

- Measures how spread out the clusters are (*lower* is better)
- Distance from each sample to centroid of its cluster
- After `fit()`, available as attribute `inertia_`
- k-means attempts to minimize the inertia when choosing clusters

```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(samples)
print(model.inertia_)
```

```
78.9408414261
```

The K-means algorithm aims to choose centroids that minimise the **inertia**, or **within-cluster sum-of-squares criterion**:

$$\sum_{i=0}^{n} \min_{\mu_j \in C} (||x_i - \mu_j||^2)$$

Inertia measures clustering quality

- Measures how spread out the clusters are (*lower* is better)
- Distance from each sample to centroid of its cluster
- After `fit()`, available as attribute `inertia_`
- k-means attempts to minimize the inertia when choosing clusters

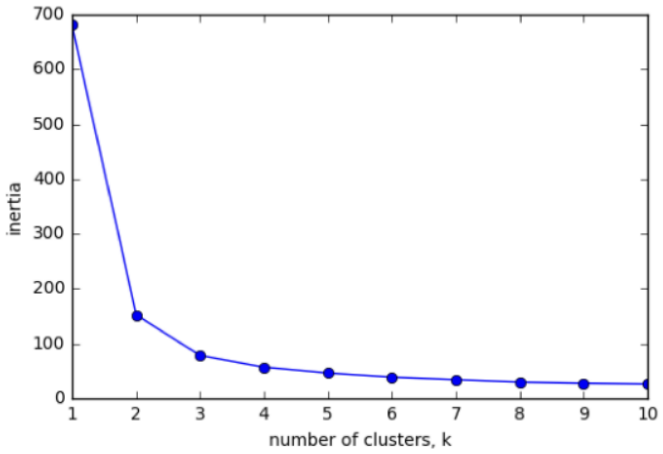
```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(samples)
print(model.inertia_)
```

```
78.9408414261
```

The number of clusters

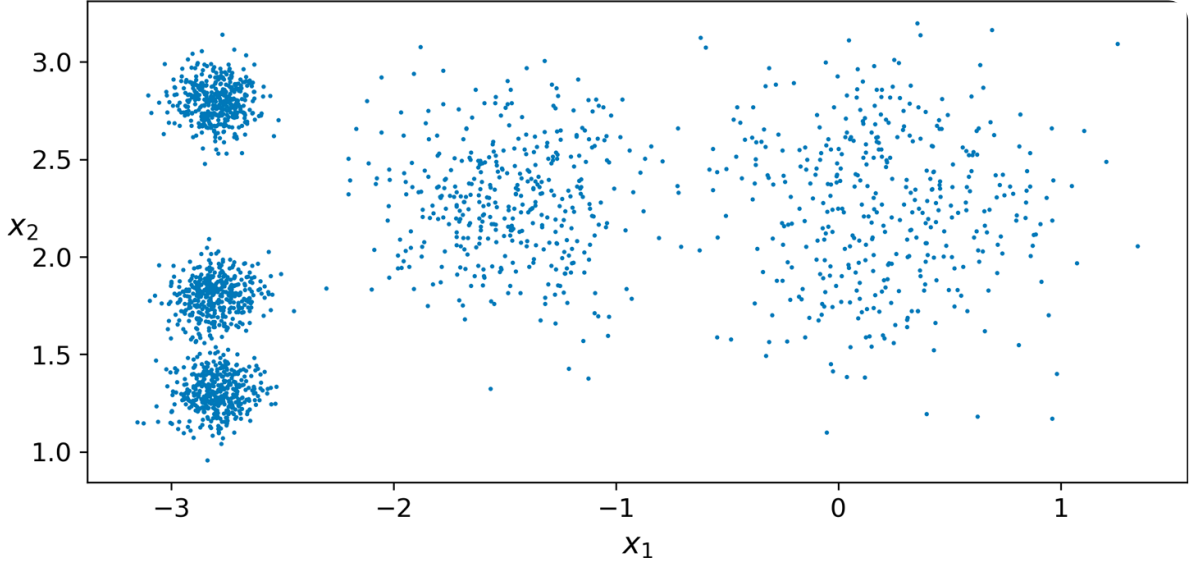
- Clusterings of the iris dataset with different numbers of clusters
- More clusters means lower inertia
- What is the best number of clusters?



- A good clustering has tight clusters (so low inertia)
- ... but not too many clusters!
- Choose an "elbow" in the inertia plot
- Where inertia begins to decrease more slowly
- E.g., for iris dataset, 3 is a good choice

K-Means

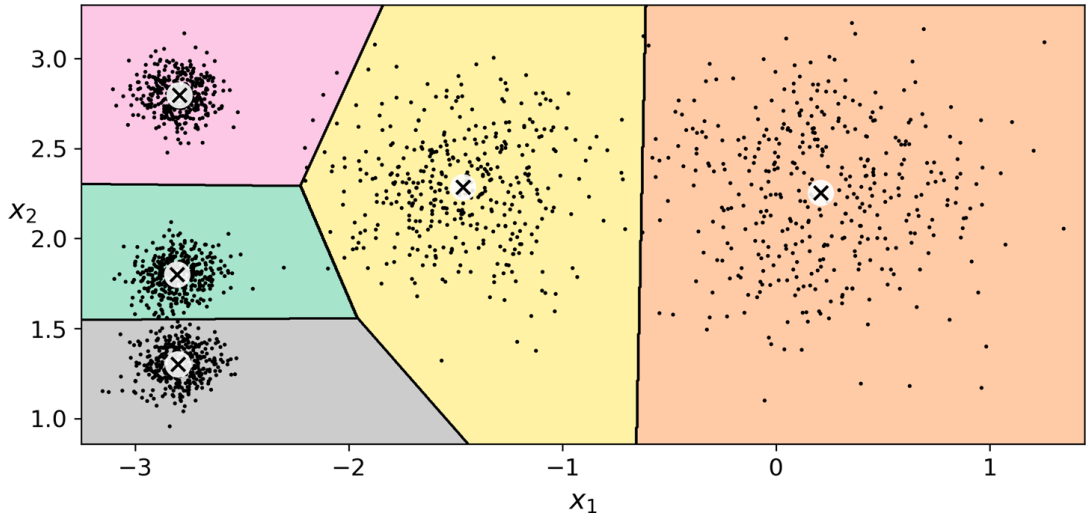
- Fast and efficient clustering
- Sometimes called Lloyd–Forgy
- K-Means++
  - smarter initialization
- Mini-batch K-Means
  - Use a portion of the full dataset at each iteration
  - Able to cluster large dataset




```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

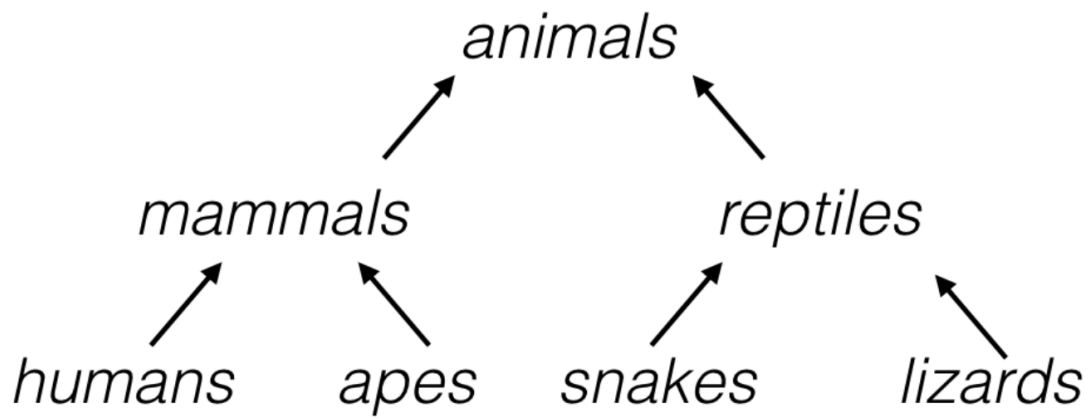
```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```



Hierarchical Tree

A hierarchy of groups

- Groups of living things can form a hierarchy
- Clusters are contained in one another



Hierarchical Clustering

As mentioned before, hierarchical clustering relies using these clustering techniques to find a hierarchy of clusters, where this hierarchy resembles a tree structure, called a dendrogram.

> Hierarchical clustering is the hierarchical decomposition of the data based on group similarities

Finding hierarchical clusters

There are two top-level methods for finding these hierarchical clusters:

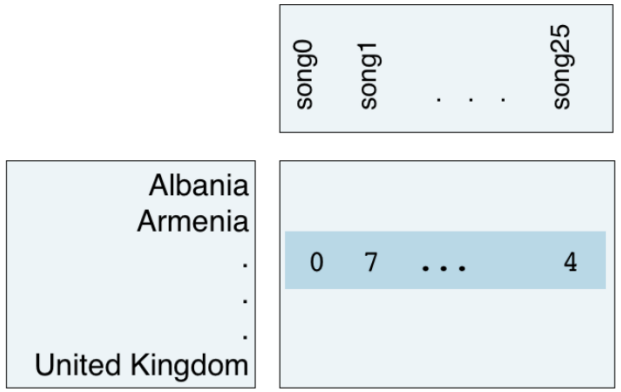
- **Agglomerative** clustering uses a *bottom-up* approach, wherein each data point starts in its own cluster. These clusters are then joined greedily, by taking the two most similar clusters together and merging them.
- **Divisive** clustering uses a *top-down* approach, wherein all data points start in the same cluster. You can then use a parametric clustering algorithm like K-Means to divide the cluster into two clusters. For each cluster, you further divide it down to two clusters until you hit the desired number of clusters.

Both of these approaches rely on constructing a similarity matrix between all of the data points, which is usually calculated by cosine or Jaccard distance.

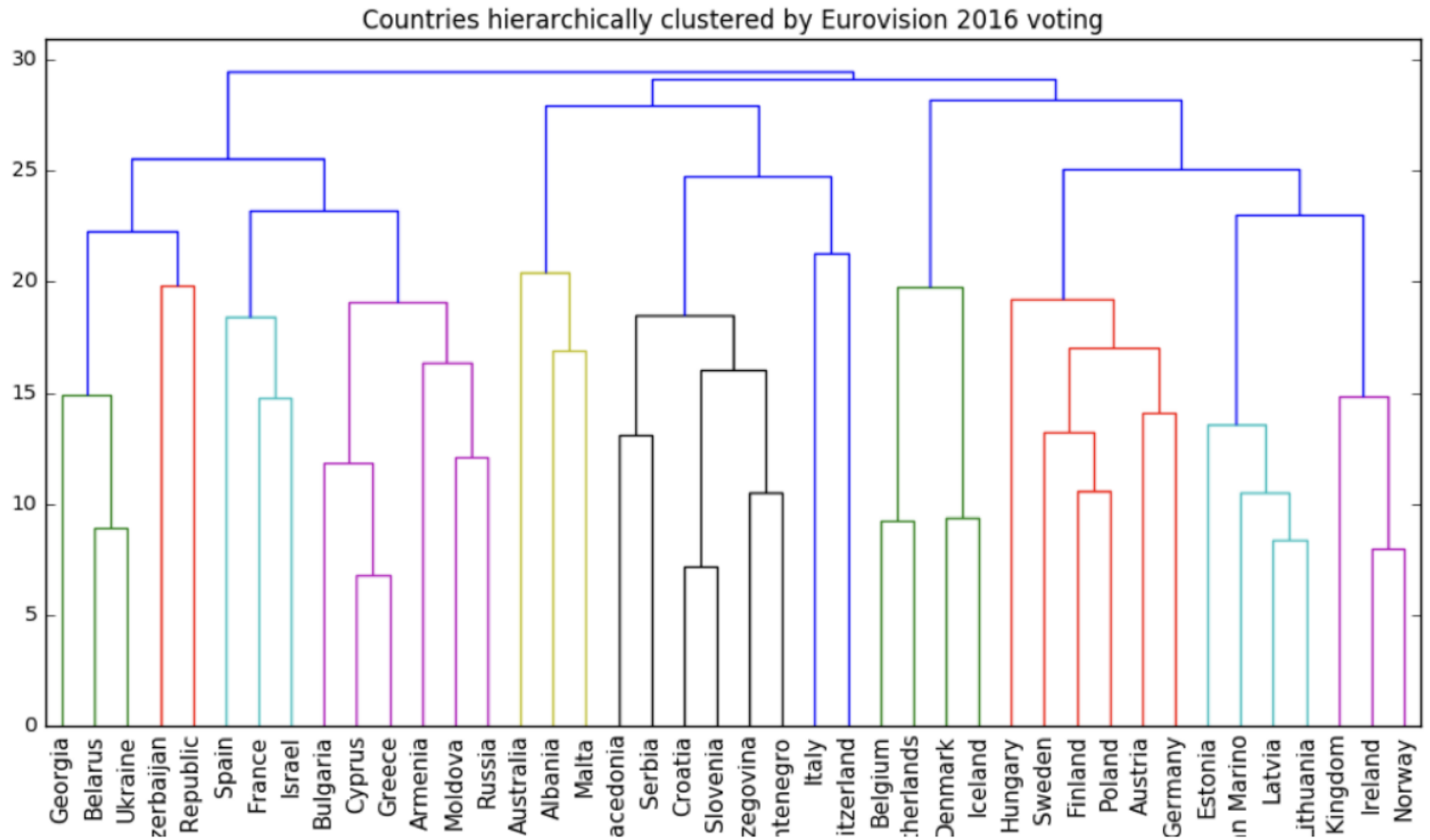
Example of bottom-up hierarchical clustering

Eurovision scoring dataset

- Countries gave scores to songs performed at the Eurovision 2016
- 2D array of scores
- Rows are countries, columns are songs



Hierarchical clustering of voting countries

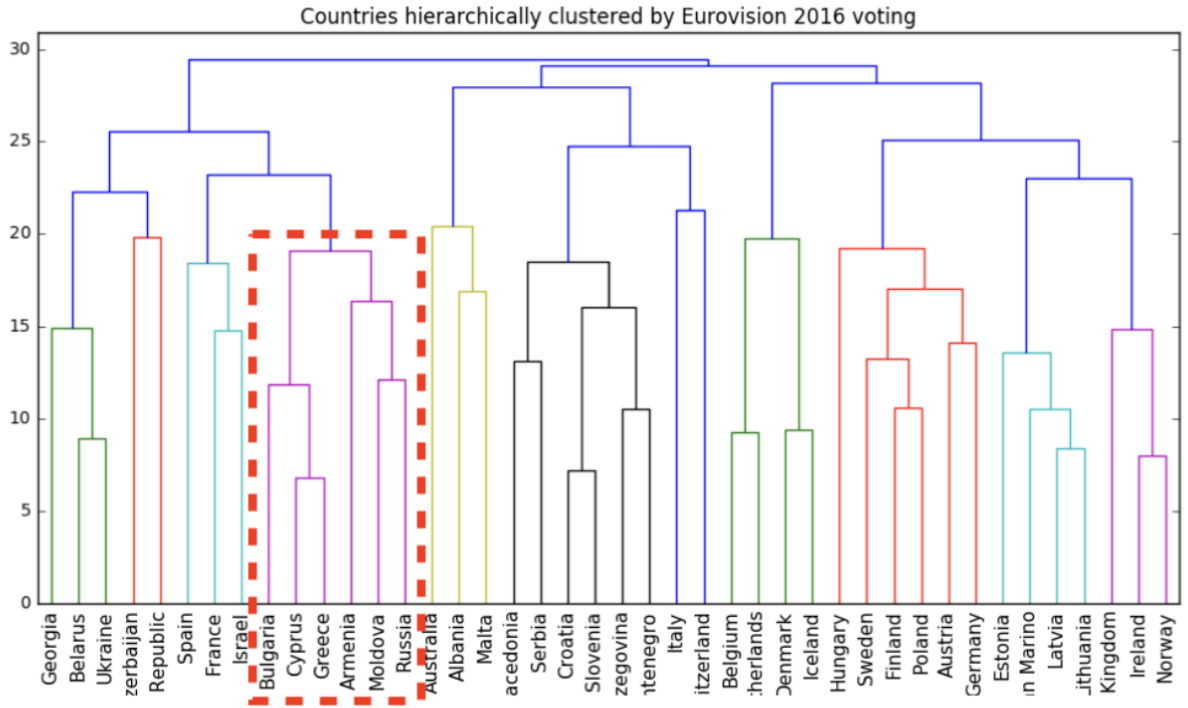


Hierarchical clustering

- Every country begins in a separate cluster
- At each step, the two closest clusters are merged
- Continue until all countries in a single cluster
- This is "agglomerative" hierarchical clustering

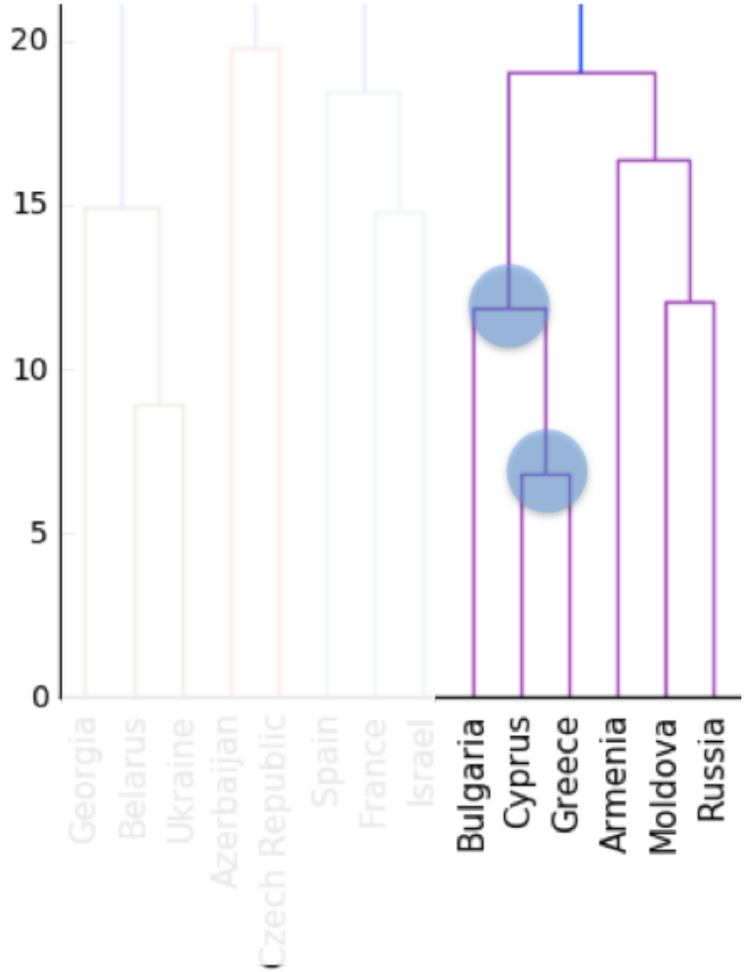
The dendrogram of a hierarchical clustering

- Read from the bottom up
- Vertical lines represent clusters



Dendrograms show cluster distances

- Height on dendrogram = distance between merging clusters
- E.g. clusters with only Cyprus and Greece had distance approx. 6
- This new cluster distance approx. 12 from cluster with only Bulgaria



Distance between clusters

- Defined by a "linkage method"
- In "complete" linkage: distance between clusters is max. distance between their samples
- Specified via method parameter, e.g. linkage(samples, method="complete")
- Different linkage method, different hierarchical clustering!

Hierarchical clustering with SciPy

- Given `samples` (the array of scores), and `country_names`

```
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram
mergings = linkage(samples, method='complete')
dendrogram(mergings,
           labels=country_names,
           leaf_rotation=90,
           leaf_font_size=6)
plt.show()
```

Distance Metrics

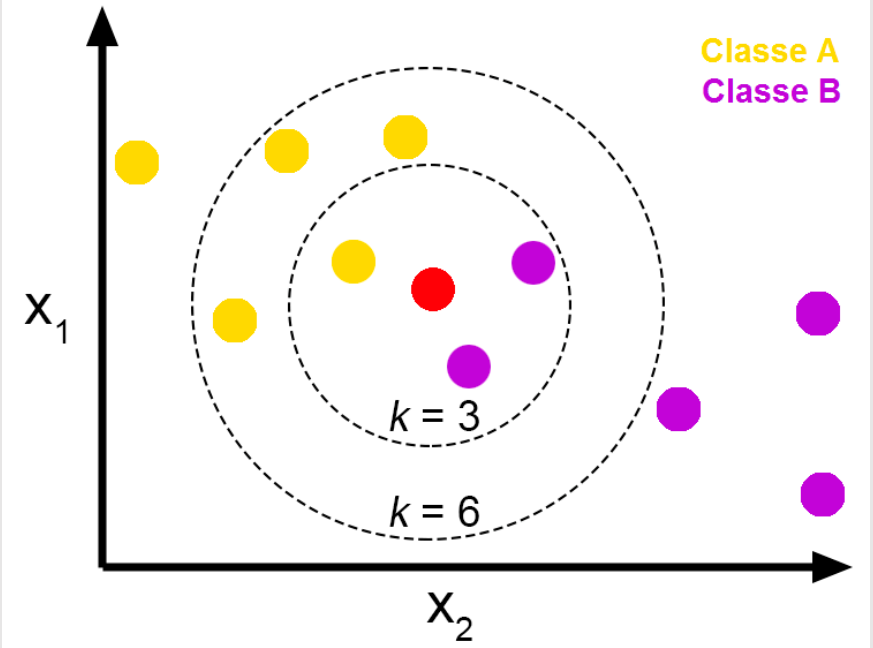
- Euclidian ($(x_i - y_i)^2$)/Manhattan ($|x_i - y_i|$) Distance: measures 3-D distance
- Minkowski Distance: measures N-D distance
- Cosine Distance: measures distance b/w embeddings
- See **SimilarityBased.ipynb** for more details

K-Nearest Neighbor: Classify Instances Based on Distances

Key Benefit: NO TRAINING NEEDED!

STEPS FOR PREDICTION

1. Receive an unclassified data;
2. Measure the distance (Euclidian, Manhattan, Minkowski or Weighted) from the new data to all others data that is already classified;
3. Gets the K(K is a parameter that you define) smaller distances (*default = 5 for sklearn*);
4. Check the list of classes had the shortest distance and count the amount of each class that appears;
5. Takes as correct class the class that appeared the most times;
6. Classifies the new data with the class that you took in step 5;

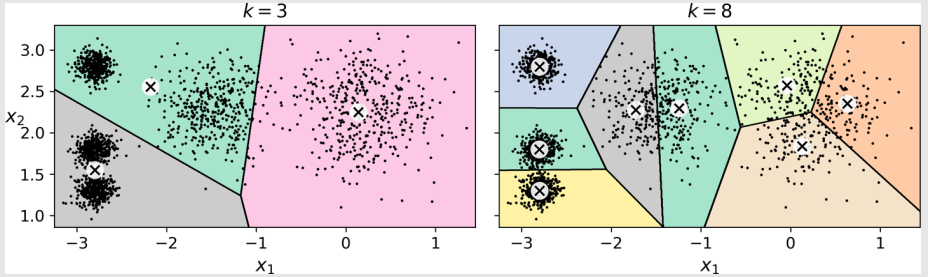


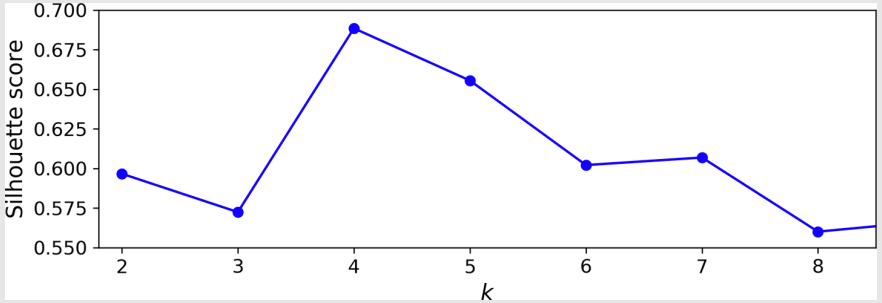
- If k = 3, red dot will be classified as **purple**
- If k = 6, red dot will be classified as **yellow**

Source:
- https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d
- https://medium.com/machinelearningalgorithms/k-nearest-neighbors-c9823dca611b
- https://medium.com/@chiragsehra42/k-nearest-neighbors-explained-easily-c26706aa5c7f

Finding Optimal Clusters

- Mostly trial-and-error
- A precise and expensive approach is by using *silhouette score*
- *The larger the better*






```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```