# **Unsupervised Learning, K-Means**

Although most of the applications of machine learning today are based on supervised learning, the vast majority of the available data is unlabeled: we have input features X, but we do not have the labels y. Suppose we want to create a system that will take a few pictures of each item on the manufacturing production line and direct which items are defective. We can easily create a system that will take pictures automatically, and this might give us thousands of pictures every day. And we can build a reasonably big dataset in just a few weeks. But, there are no labels, so, if we want to train a regular binary classifier that will predict whether an item is defective or normal. There are several tasks and algorithms to solve unsupervised learning problems:

Clustering: The goal is to group similar instances into clusters. Clustering is a great tool for data analysis, customer segmentation, recommendation systems, search engines, image segmentation, sem-supervised learning, dimensionality reduction, and more.

Anomaly direction: The objective is to learn what ‘normal’ data looks like, and then use that to detect abnormal instances, such as defective items on the production line or a new trend in a time series.

Density estimation: This is the task of estimating the probability density function of a random process that generates a dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

Dimensionality reduction: Dimensionality reduction is a technique used in machine learning and data analysis to reduce the number of input variables or features in a dataset. The goal of dimensionality reduction is to simplify the dataset while preserving as much relevant information as possible. High-dimensional datasets often suffer from the curse of dimensionality, leading to increased computational complexity, overfitting, and difficulty in visualization and interpretation.

Consider that we have visualized unlabeled data and can see 5 blobs of instances. The K-Means algorithm is a simple algorithm capable of clustering this kind of data very quickly and efficiently, with just a few iterators. So, let’s see how we need to train a K-Means clustered on a certain dataset. It will try to find each blob’s center and assign each instance to the closest blob. We must specify the number of clusters k the algorithm must find. In this example, it is pretty obvious from the data that k should be set to 5, but generally, it is not that easy. Each instance will assigned to one of the five clusters. In the context of clustering, an instance’s label is the index of the cluster that this instance gets assigned to by the algorithm. The vast majority of the instances will be assigned to the appropriate cluster, but a few instances will probably be mislabeled. Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid. Instead of assigning each instance to a single cluster, which is called hard clustering, it can be useful to give each instance a score per cluster, which is called soft clustering. The score can be the distance between the instance and the centroid.

So, how does the algorithm work? We begin the k-means algorithm by picking K, and randomly assigning a roughly equal number of observations to each of the K clusters. Then K-means consists of two major steps that attempt to minimize the sum of WSSDs over all the clusters

Center update: Compute the center of each cluster.

Label update: Reassign each data point to the cluster with the nearest center.

These two steps are repeated until the cluster assignments no longer change. Unlike the classification and regression models we studied in previous chapters, K-means can get “stuck” in a bad solution. This looks like a relatively bad clustering of the data, but K-means cannot improve it. To solve this problem when clustering data using K-means, we should randomly re-initialize the labels a few times, run K-means for each initialization, and pick the clustering that has the lowest final total WSSD. To cluster data using K-means, we also have to pick the number of clusters, K. But unlike in classification, we have no response variable and cannot perform cross-validation with some measure of model prediction error. Further, if K is chosen too small, then multiple clusters get grouped; if K is too large, then clusters get subdivided. In both cases, we will potentially miss interesting structures in the data. If we set K less than 3, then the clustering merges separate groups of data; this causes a large total WSSD since the cluster center (denoted by large shapes with black outlines) is not close to any of the data in the cluster. On the other hand, if we set K greater than 3, the clustering subdivides subgroups of data; this does indeed still decrease the total WSSD, but by only a diminishing amount. If we plot the total WSSD versus the number of clusters, we see that the decrease in total WSSD levels off (or forms an “elbow shape”) when we reach roughly the right number of clusters

Let X = x1,x2 , .... , xn be the dataset consisting of n data points, where each data point xi belongs to a d-dimensional space. Let C = c1,c2 , .... , ck denote the set of centroids, where k is the number of clusters. The objective of the k-means algorithm is to minimize the within-cluster sum of squares (WCSS), given by:

WCSS = i=1kxCix - i2

Where Ci is the ith cluster, i is the centroid of Ci, and denotes the Euclidean distance.

Implementation:

from sklearn.datasets import make\_blobs

import matplotlib.pyplot as plt

x, y = make\_blobs(n\_samples=500, centers=3, random\_state=70)

plt.figure(figsize=(8, 6))

plt.scatter(x[:, 0], x[:, 1], c=y, cmap='viridis')

plt.show()

k\_means = KMeans(n\_clusters=3, init= "k-means++",random\_state=0)

k\_means.fit(x)

y\_pred = k\_means.predict(x)

centroids = k\_means.cluster\_centers\_

import matplotlib.pyplot as plt

plt.scatter(x[:,0],x[:,1], c = y\_pred)

plt.scatter(centroids[:,0],centroids[:,1], s = 500, c = 'lightblue')

wcss = []

for i in range(1, 11):

k\_means = KMeans(n\_clusters=i, init= "k-means++",random\_state=0)

k\_means.fit(x)

y\_pred = k\_means.predict(x)

centroids = k\_means.cluster\_centers\_

wcss\_i = k\_means.inertia\_

wcss.append(wcss\_i)

plt.scatter(x[:,0],x[:,1], c = y\_pred)

plt.scatter(centroids[:,0],centroids[:,1], s = 500, c = 'lightblue')

plt.show()

plt.plot(range(1,11),wcss)