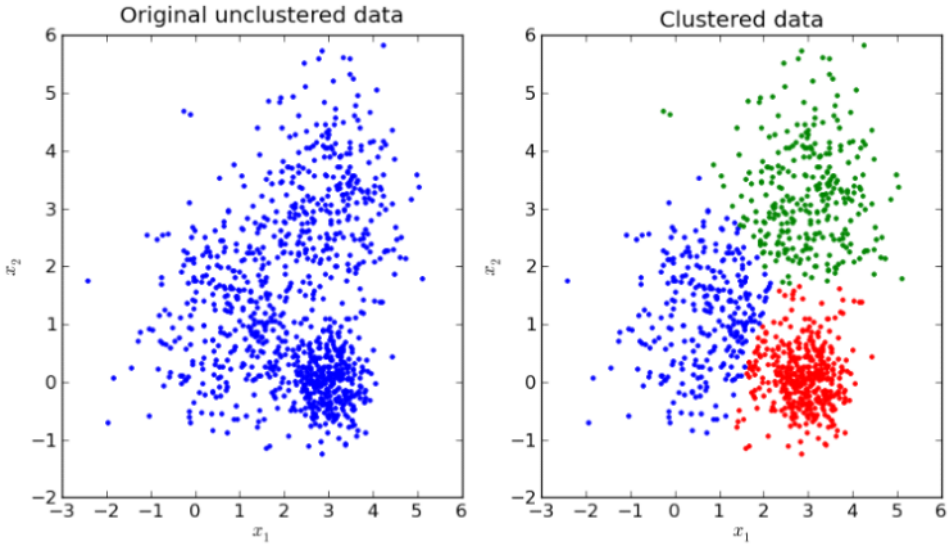
13 March 2019
14:58

# K-Means Clustering in Python

Clustering is a type of Unsupervised learning. This is very often used when you don't have labeled data. K-Means Clustering is one of the popular clustering algorithm. The goal of this algorithm is to find groups(clusters) in the given data. In the below lines we will implement K-Means algorithm using Python from scratch.

## K-Means Clustering

K-Means is a very simple algorithm which clusters the data into *K* number of clusters. The following image is an example of K-Means Clustering.




**Use Cases:**

K-Means is widely used for many applications.

- Image Segmentation
- Clustering Gene Segmentation Data
- News Article Clustering
- Clustering Languages
- Species Clustering
- Anomaly Detection

Parameters of "scikit-learn"

| Parameters | **n_clusters :** ***int, optional, default: 8***<br>The number of clusters to form as well as the number of centroids to generate.<br><br>**init :** ***{'k-means++', 'random' or an ndarray}***<br>Method for initialization, defaults to 'k-means++':<br>'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.<br>'random': choose k observations (rows) at random from data for the initial centroids.<br>If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the |
|---|---|

initial centers.

**n_init :** ***int, default: 10***
Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

**max_iter :** ***int, default: 300***
Maximum number of iterations of the k-means algorithm for a single run.

**tol :** ***float, default: 1e-4***
Relative tolerance with regards to inertia to declare convergence

**precompute_distances :** ***{'auto', True, False}***
Precompute distances (faster but takes more memory).
'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.
True : always precompute distances
False : never precompute distances

**verbose :** ***int, default 0***
Verbosity mode.

**random_state :** ***int, RandomState instance or None (default)***
Determines random number generation for centroid initialization. Use an int to make the randomness deterministic.

**copy_x :** ***boolean, optional***
When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified, ensuring X is C-contiguous. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean, in this case it will also not ensure that data is C-contiguous which may cause a significant slowdown.

**n_jobs :** ***int or None, optional (default=None)***
The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

**algorithm :** ***"auto", "full" or "elkan", default="auto"***
K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.