K-Means Clustering

Learn about the K-Means clustering algorithm and how it works.

Chapter Goals:

- Learn about K-means clustering and how it works
- Understand why mini-batch clustering is used for large datasets

A. K-means algorithm

The idea behind clustering data is pretty simple: partition a dataset into groups of similar data observations. How we go about finding these clusters is a bit more complex, since there are a number of different methods for clustering datasets.

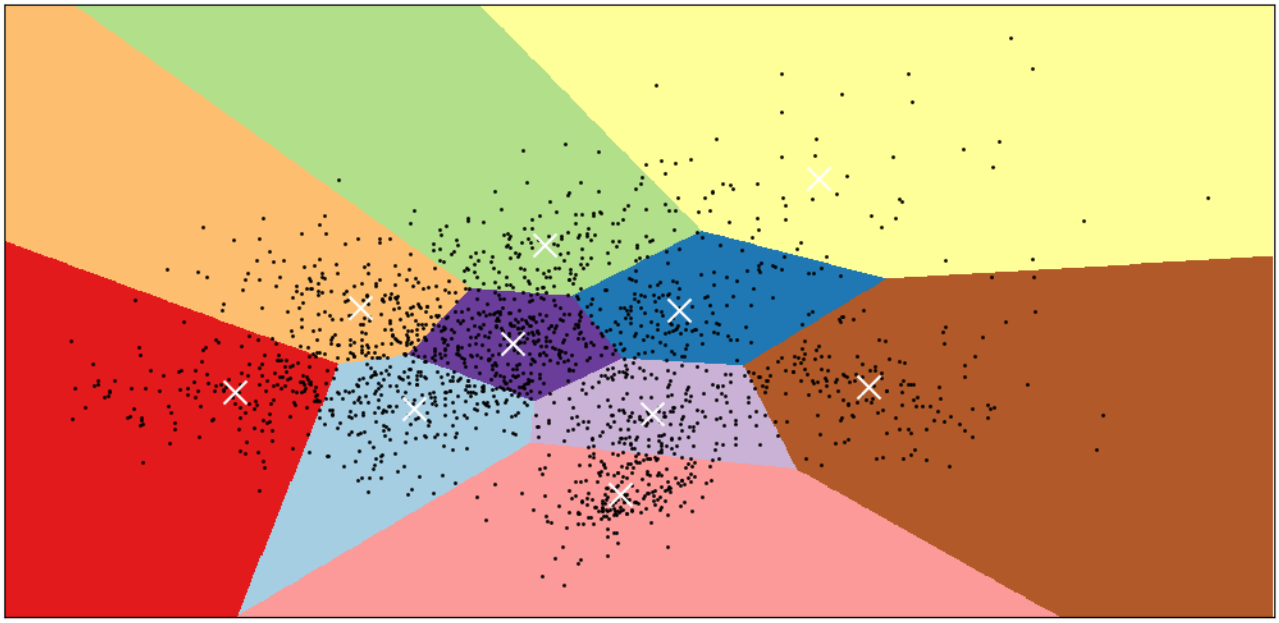
The most well-known clustering method is K-means clustering. The K-means clustering algorithm will separate the data into *K* clusters (the number of clusters is chosen by the user) using cluster means, also known as *centroids*.

These centroids represent the "centers" of each cluster. Specifically, a cluster's centroid is equal to the average of all the data observations within the cluster.

```
cluster = np.array([
  [ 1.2, 0.6],
  [ 2.4, 0.8],
  [-1.6, 1.4],
  [ 0. , 1.2]])
print('Cluster:\n{}\n'.format(repr(cluster)))

centroid = cluster.mean(axis=0)
print('Centroid:\n{}\n'.format(repr(centroid)))
```

The K-means clustering algorithm is an iterative process. Each iteration, the algorithm will assign each data observation to the cluster with the closest centroid to the observation (using the regular distance metric).

Then it updates each centroid to be equal to the new average of the data observations in the cluster. Note that at the beginning of the algorithm, the cluster centroids are either randomly initialized or (better) initialized using the K-means++ algorithm. The clustering process stops when there are no more changes in cluster assignment for any data observation.



An example of K-means clustering on a dataset with 10 clusters chosen (K = 10). Clusters are distinguished by color. The white crosses represent the centroids of each cluster.

In scikit-learn, K-means clustering is implemented using the `KMeans` object (part of the `cluster` module).

The code below demonstrates how to use the `KMeans` object (with 3 clusters).

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3)
# predefined data
kmeans.fit(data)

# cluster assignments
print('{}\n'.format(repr(kmeans.labels_)))

# centroids
print('{}\n'.format(repr(kmeans.cluster_centers_)))

new_obs = np.array([
  [5.1, 3.2, 1.7, 1.9],
  [6.9, 3.2, 5.3, 2.2]])
# predict clusters
print('{}\n'.format(repr(kmeans.predict(new_obs))))
```

The `KMeans` object uses K-means++ centroid initialization by default. The `n_clusters` keyword argument lets us set the number of clusters we want. In the example above, we applied clustering to `data` using 3 clusters.

The `labels_` attribute of the object tells us the final cluster assignments for each data observation, and the `cluster_centers_` attribute represents the final centroids. We use the `predict` function to assign new data observations to one of the clusters.

B. Mini-batch clustering

When working with very large datasets, regular K-means clustering can be quite slow. To reduce the computation time, we can perform *mini-batch* K-means clustering, which is just regular K-means clustering applied to randomly sampled subsets of the data (mini-batches) at a time.

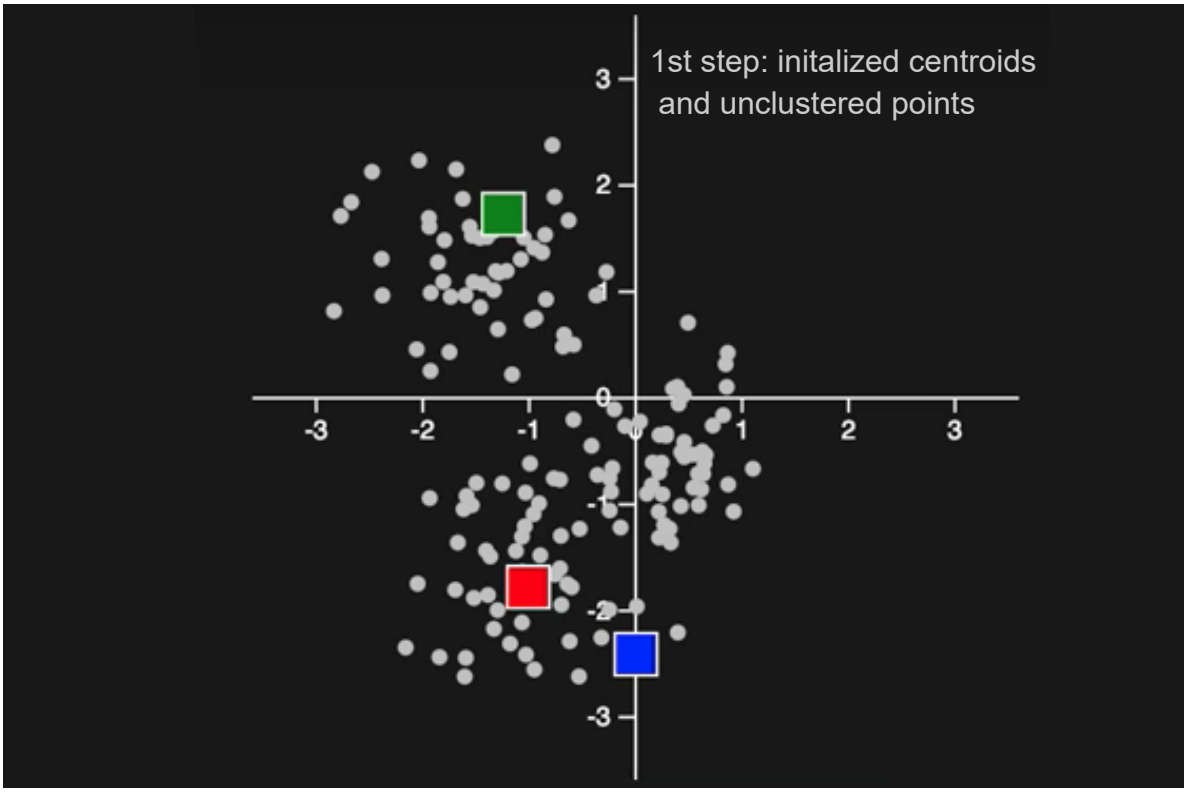
There is a trade-off in using mini-batch clustering, as the results may not be as good as regular K-means clustering. However, in practice the difference in quality is negligible, so mini-batch clustering is usually the choice when dealing with large datasets.

In scikit-learn, mini-batch K-means clustering is implemented using the `MiniBatchKMeans` object (also part of the `cluster` module). It is used in the same way as the regular `KMeans` object, with an additional `batch_size` keyword argument during initialization that allows us to specify the size of each mini-batch.

```
from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(n_clusters=3, batch_size=10)
# predefined data
kmeans.fit(data)

# cluster assignments
print('{}\n'.format(repr(kmeans.labels_)))

# centroids
print('{}\n'.format(repr(kmeans.cluster_centers_)))

new_obs = np.array([
  [5.1, 3.2, 1.7, 1.9],
  [6.9, 3.2, 5.3, 2.2]])
# predict clusters
```

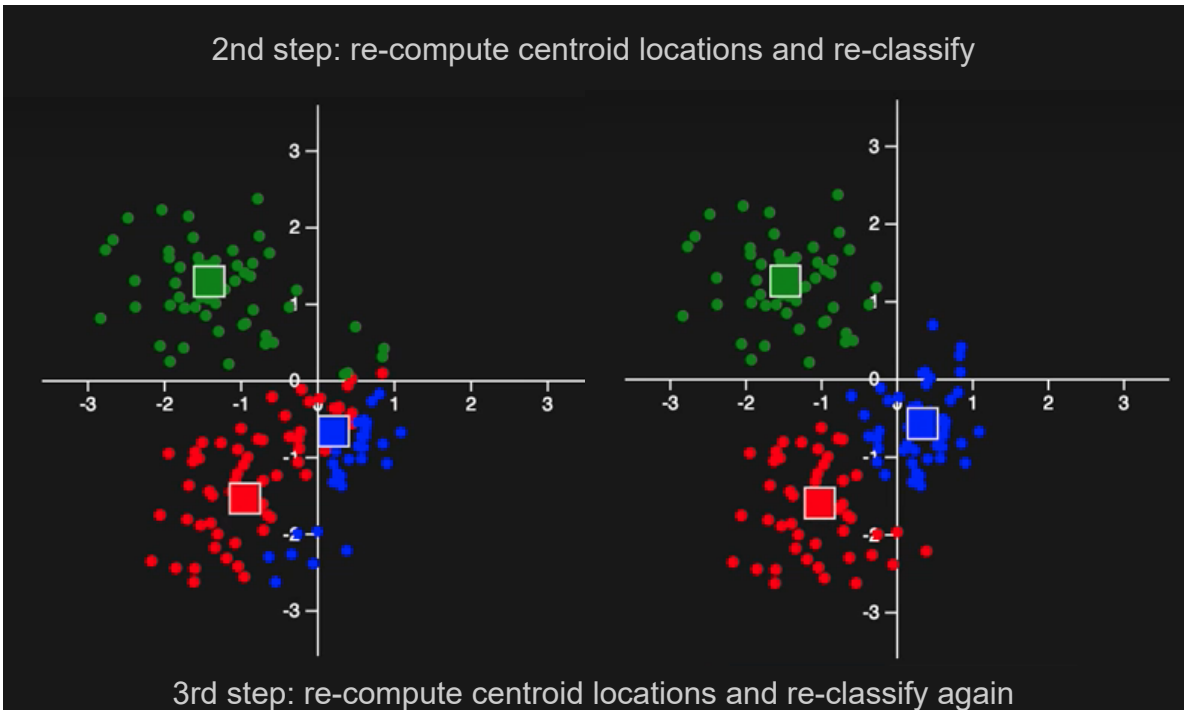
```
print('{}\n'.format(repr(kmeans.predict(new_obs))))
```

Note that the clusterings can have different permutations, i.e. different cluster labelings (0, 1, 2 vs. 1, 2, 0). Otherwise, the cluster assignments for both `KMeans` and `MiniBatchKMeans` should be relatively the same.



Example of the K-Means algorithm's multiple steps.

1 of 2



The most noticeable difference is which points are in the red and blue clusters

Time to Code!

The coding exercise for this chapter will be to complete the `kmeans_clustering` function, which will use either `KMeans` or `MiniBatchKMeans` for clustering `data`.

Which object we use for clustering depends on the value of `batch_size`.

If `batch_size` is `None`, set `kmeans` equal to `KMeans`, initialized with `n_clusters` for the `n_clusters` argument. Otherwise, set `kmeans` equal to `MiniBatchKMeans`, initialized with `n_clusters` for the `n_clusters` argument and `batch_size` for the `batch_size` argument.

After setting up `kmeans`, we fit it on the `data` and return it.

Call `kmeans.fit` with `data` as the only argument. Then return `kmeans`.

```
def kmeans_clustering(data, n_clusters, batch_size):
  # CODE HERE
  pass
```