K Means Clustering

`k-means clustering` is a method of `vector quantization`, originally from signal processing, that 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`. This results in a partitioning of the `data` space into `Voronoi cells`.(Too many fancy words mumbo jumbo, right? Don't worry, I will explain everything in a simple way)

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Introduction

What is K-means Clustering?

`K-means` is an `unsupervised learning` algorithm used to solve `clustering` problems on `unlabeled` data.

The main idea of `K-means clustering` is very simple: it takes the `input data` and tries to divide it into multiple `groups` (called `clusters`). But how many `groups` should there be? That's where the `K` comes in.

The `K` in `K-means` represents the number of `clusters` we want to divide our `data` into. We manually choose this value and give it to the algorithm.

Once the value of `k` is provided, the algorithm creates `k` different `clusters`. Each `cluster` has a `centroid`, which represents the `center` of that cluster.

The algorithm then assigns each `data point` to the `closest centroid`. Here, `closest` means the centroid with the `minimum distance` from that data point (usually measured using `Euclidean distance`).

After assigning all data points, the algorithm recalculates the `centroid` of each cluster by taking the `mean` of all the data points inside that cluster. This newly calculated `mean` becomes the updated `centroid`.

This process of:

- assigning data points to the `closest centroid`
- recalculating the `mean` to update centroids

is repeated again and again until the `centroids` stop changing (or change very little).

Once this happens, the algorithm stops, and we get the final `clusters`.

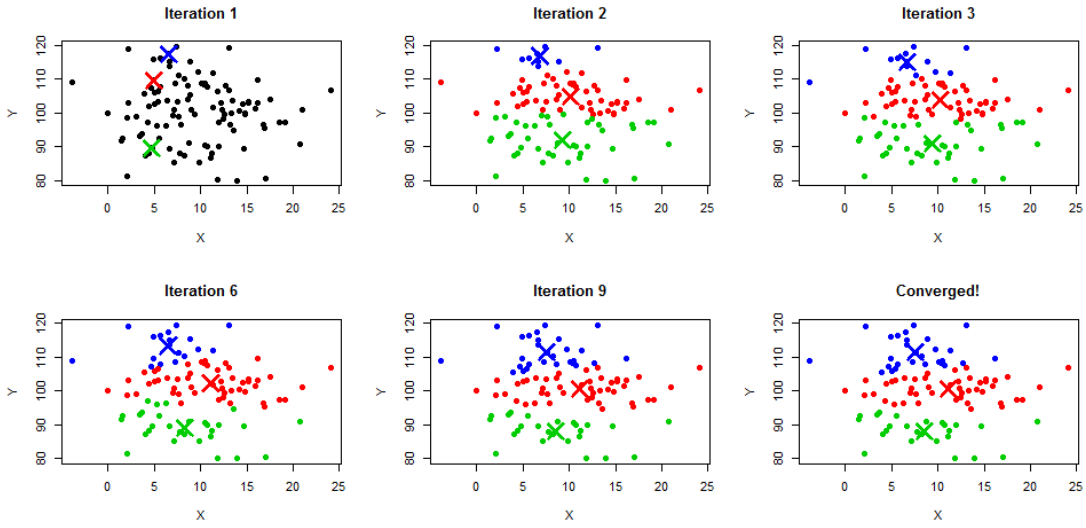
So in short, `K-means` is an `iterative algorithm` that continuously updates `centroids` and reassigns data points until a stable clustering is achieved.

How does K-means Clustering work?

The working process of the `K-means` algorithm can be broken down into the following steps:

1. **Initialize centroids** First, we choose `k` initial `centroids`. This can be done in multiple ways:
    - randomly selecting `k` data points from the dataset
    - randomly generating `k` points in the feature space (In practice, smarter methods like `k-means++` are often used.)
2. **Assign data points to the closest centroid** Each `data point` is assigned to the `centroid` that has the `least distance` from it.
3. **Recalculate centroids** For each `cluster`, we calculate the `mean` of all assigned data points. This `mean` becomes the new `centroid` of that cluster.
4. **Repeat the process** Steps 2 and 3 are repeated:
    - reassign data points
    - update centroids
5. **Stop condition** The algorithm stops when:
    - the `centroids` no longer change, or
    - the change is very small, or
    - a maximum number of iterations is reached

After stopping, the final `clusters` represent the output of the `K-means` algorithm.



How `K-Means` Clustering Works (Step by Step)

Step 1: Choose the number of clusters (`k`)

Before the algorithm starts, you must decide how many clusters you want.

For example:

- `k = 2` → split data into 2 groups
- `k = 3` → split data into 3 groups

At this stage, `k` is just a guess. Later, we'll see how the `elbow method` helps choose a good value.

Step 2: Initialize cluster centroids randomly

Once `k` is chosen, the algorithm randomly places `k` points in the feature space.

These points are called `centroids` .

Important:

- Centroids are **not real data points**
- They are just coordinates representing the center of a cluster

Because this step is random, different runs can produce different results.

Step 3: Assign each data point to the nearest centroid

For each data point:

1. Compute the distance to each centroid (usually `Euclidean distance`)
2. Assign the point to the cluster whose centroid is closest

After this step:

- Every data point belongs to **exactly one cluster**
- Clusters are formed based on proximity

Step 4: Update the centroids

Now that clusters are formed:

- Compute the `mean` of all points in each cluster
- Move the centroid to this new mean position

This is why it's called `K-Means` :

- `K` clusters
- Centroids are the `mean` of cluster points

Step 5: Repeat assignment and update steps

Steps 3 and 4 are repeated iteratively:

- Assign points to the nearest centroid
- Recalculate centroids based on current assignments

This continues until:

- centroids no longer move significantly
- or cluster assignments stop changing
- or a maximum number of iterations is reached

At this point, the algorithm is said to have `converged`.

What does K-Means optimize? (`SSE`)

K-Means is not just moving points randomly. It is minimizing a specific objective function called `SSE`.

`SSE` measures how compact the clusters are.

Formally:

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} |x - \mu_i|^2$$

Where:

- C_i is the i-th cluster
- μ_i is the centroid of cluster `C_i`
- x is a data point

the intuition is:

- Measure distance between each point and its centroid
- Square the distance
- Sum it for all points and all clusters

Lower `SSE` means:

- tighter clusters
- points are closer to their centroids

Each iteration of K-Means **guarantees that SSE decreases or stays the same**.

Now you might think, why does SSE always decrease?

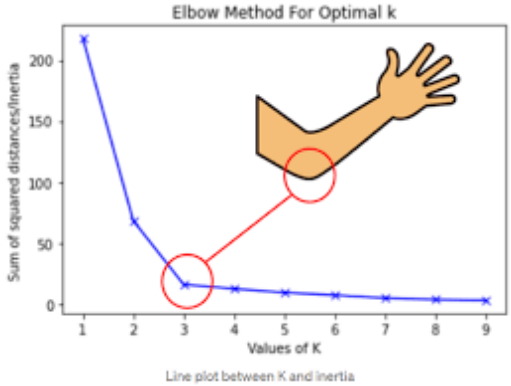
- Assignment step reduces distance to nearest centroid
- Update step moves centroid to minimize squared distance

This is why K-Means converges.

However:

- it may converge to a **local minimum**
- not always the global best clustering

Choosing the right `k`: The `Elbow Method`



Line plot between K and Inertia

Choosing `k` arbitrarily is risky.

The `elbow method` helps estimate a good value.

1. Run K-Means for different values of `k` (e.g. `k = 1` to `k = 10`)
2. For each `k`, compute the `SSE`
3. Plot `k` vs `SSE`

You would quickly notice that the SSE decreases as `k` increases and at some point, improvement slows down sharply

That point looks like an **elbow** in the graph.

- Small `k` → clusters are too broad → high SSE
- Large `k` → clusters become tighter → lower SSE
- After a certain `k`, adding more clusters gives diminishing returns

The elbow point balances:

- simplicity
- compactness

But there are some limitations you have to keep in mind, k means algorithm is,

- Sensitive to initial centroid placement
- Requires choosing `k` beforehand
- Works best with spherical, equally sized clusters
- Sensitive to feature scaling

> K-Means works by repeatedly assigning points to the nearest centroid and updating centroids to minimize the sum of squared distances, gradually forming compact and well-separated clusters.

For better understanding of `k means clustering` you can watch this video.

Now, let's see how we can make clusters using `sklearn` and use `k means algorithm` to actually cluster the data.

Implementing K means clustering

As I said earlier, `k means algorithm` doesn't require any labels for the data. It only requires the data itself. But for the purposes of this article, we will use labeled data. And in this case I'll show you how you can make `clustered` datapoints using sklearn.

It's pretty simple.

We can just use `make_blobs` function from `scikit-learn` library to make blobs of data.

In [11]:
```
from sklearn.datasets import make_blobs

data = make_blobs(n_samples=200, n_features=2, centers=4, cluster_std=2, r
```

Let me explain what I just did in the code above.

In the make blob function, we can specify the `number of samples(n_samples)`, the `number of features(n_features)`, the `number of clusters(centers)`, the `standard deviation of the clusters(cluster_std)`, and the `random seed(random_state)`.

> `cluster_std` is the standard deviation of the clusters. The higher the value, the more spread out the clusters are.

`make_blob` will return a `2D` `array` that represents the `features` of the `data` and another `array` that represents the `cluster` of each `data` point.

So, we can separate the `data` and the `cluster`,

In [12]:
```
features = data[0]
features.shape
```

Out[12]:
```
(200, 2)
```

We have a feature set of `200` `data` points with `2` `features`.

Let's get the `cluster` of each `data` point,

In [13]:
```
clusters = data[1]
clusters.shape
```

Out[13]:
```
(200,)
```

The `cluster` of each `data` point is a `1D` `array` of `200` `data` points.

Now, I'll turn this into a `dataframe` to keep things easy.

Let's visualize the `data` and the `cluster`.

In [14]:
```
import pandas as pd

df = pd.DataFrame(features, columns=['feature1', 'feature2'])
df['cluster'] = clusters
df.head()
```

Out[14]:

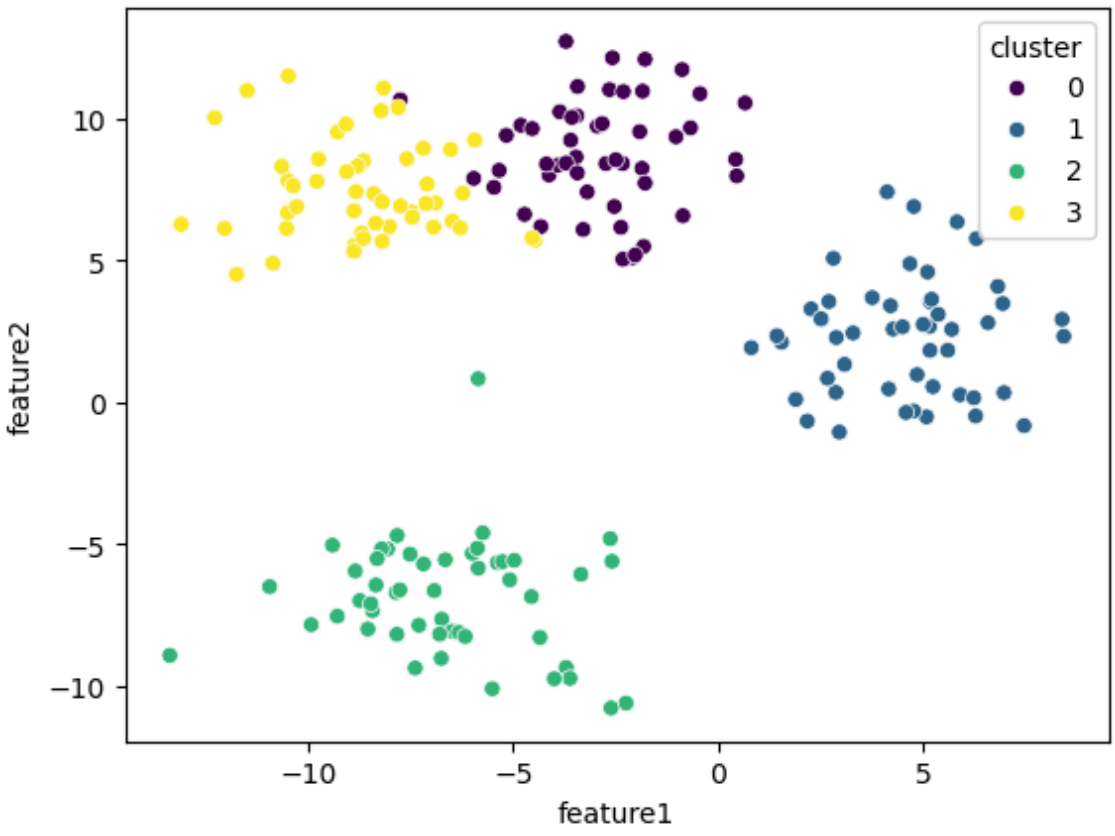
	feature1	feature2	cluster
0	8.371428	2.920836	1
1	-7.748688	10.658091	0
2	-8.862821	5.529014	3
3	-11.734496	4.508595	3
4	-8.058357	-5.180905	2

looks good. Let's plot the `data` and the `cluster`.

In [15]:
```
import seaborn as sns

sns.scatterplot(data=df, x='feature1', y='feature2', hue='cluster', palett
```

Out[15]:
```
<Axes: xlabel='feature1', ylabel='feature2'>
```



Well, the cluster looks fairly random. Let's try to cluster the data using `K-means` algorithm.

In [16]:
```
from sklearn.cluster import KMeans
```

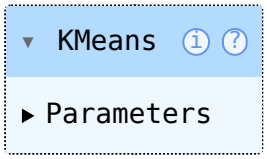
```
kmeans = KMeans(n_clusters=4)
```

Now, we fit the data to the `K-means` algorithm.

> I've set the `n_clusters` to `4` because I know that there are 4 clusters in the data. I'll show you how to find the number of clusters later in this article

In [17]:
```
kmeans.fit(features)
```

Out[17]:



The model is trained and we can now get the `cluster` centers and the `labels` of the `data`.

In [18]:
```
pred_clusters = kmeans.cluster_centers_
print(pred_clusters)

labels = kmeans.labels_
print(labels)
```

```
[[ 4.52827469  2.31311912]
 [-8.75330277  7.4646063 ]
 [-6.64425912 -7.05642342]
 [-2.86089586  8.66203102]]
[0 1 1 1 2 2 0 1 0 2 2 0 0 2 2 2 3 1 2 2 2 2 1 3 1 3 3 2 3 0 2 2 1 1 3 0 1
 0 1 3 2 3 2 2 1 0 0 2 0 3 3 3 1 0 3 3 2 2 3 0 1 0 2 1 1 2 0 3 1 3 3 1 3 2
 0 2 0 3 1 3 3 0 2 1 1 1 1 3 0 1 2 3 0 0 0 1 3 0 2 3 1 1 3 2 3 0 1 2 2 1 0
 2 3 1 3 1 3 1 3 3 3 2 0 1 1 0 3 0 0 3 2 2 3 1 1 0 2 2 3 2 0 3 1 0 0 3 0 1
 2 2 3 1 0 1 2 1 1 0 0 0 3 0 0 1 3 2 0 0 2 0 1 3 2 2 0 2 0 3 3 2 3 2 1 1 1
 3 0 0 0 3 3 2 1 1 3 1 0 3 2 0]
```

Now, we visualize if we have correctly clustered the data like we did before.

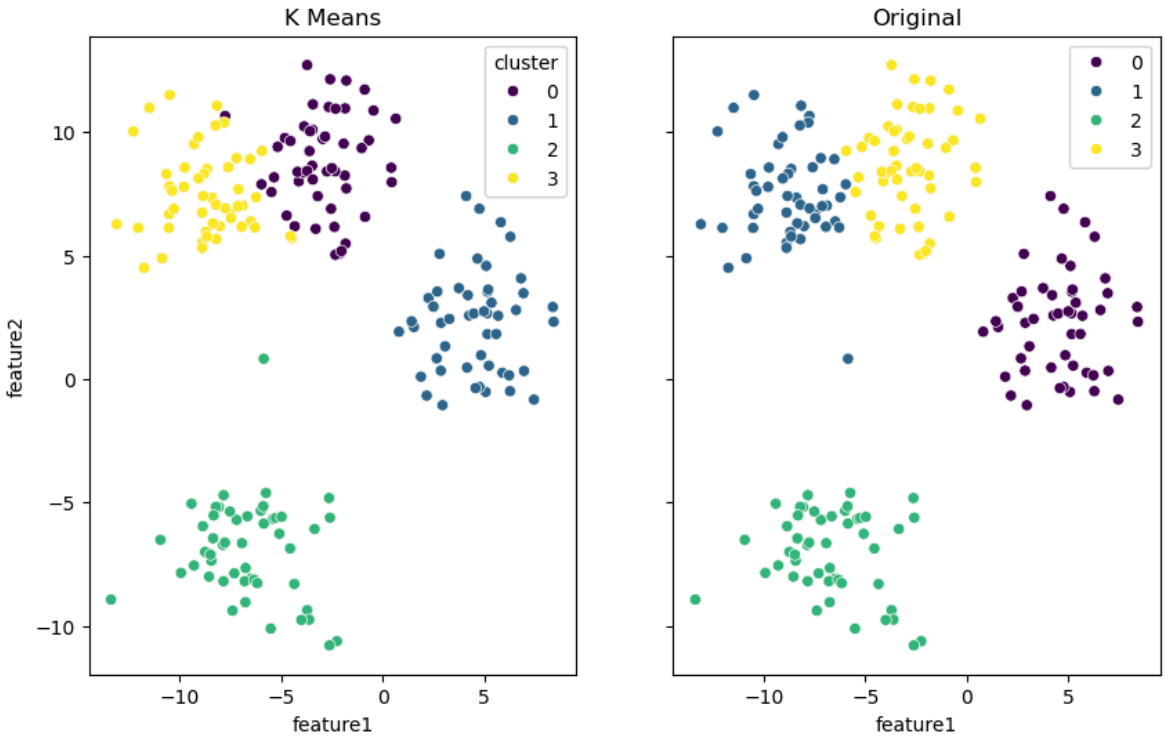
In [19]:
```
from matplotlib import pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(10,6))

ax1.set_title('K Means')
sns.scatterplot(data=df, x='feature1', y='feature2', hue='cluster', palett

ax2.set_title('Original')
sns.scatterplot(data=df, x='feature1', y='feature2', hue=labels, palette='
```

Out[19]:
```
<Axes: title={'center': 'Original'}, xlabel='feature1', ylabel='feature
2'>
```



> the colores don't mean anything. They are just for visualization purposes.

SO, as you can see the clusters are almost the same as before. The `K-means` algorithm has done a good job of clustering the `data`.

Adn sometimes this algorithm will fix the clusters in a different order. But the main thing is that the `K-means` algorithm has clustered the `data` into `4` clusters and it has done a good job of clustering the `data`.

Now let's say we don't know the number of clusters in the data. We can use the `Elbow method` to find the number of clusters in the data.

Elbow method

The `Elbow method` is a method that is used to find the number of clusters in the data. It is a simple method that is used to find the number of clusters in the data.

As I told you earlier we can use `SSE(Sum of Squared Errors)` to find the number of clusters in the data.

Good for use the `kmeans` after training does give the `SSE` value via the `inertia` attribute.

So, what we can do is select a range of `k` values and then calculate the `SSE` for each value and then plot the `SSE` values against the `k` values.

And find the best `k` for the data.

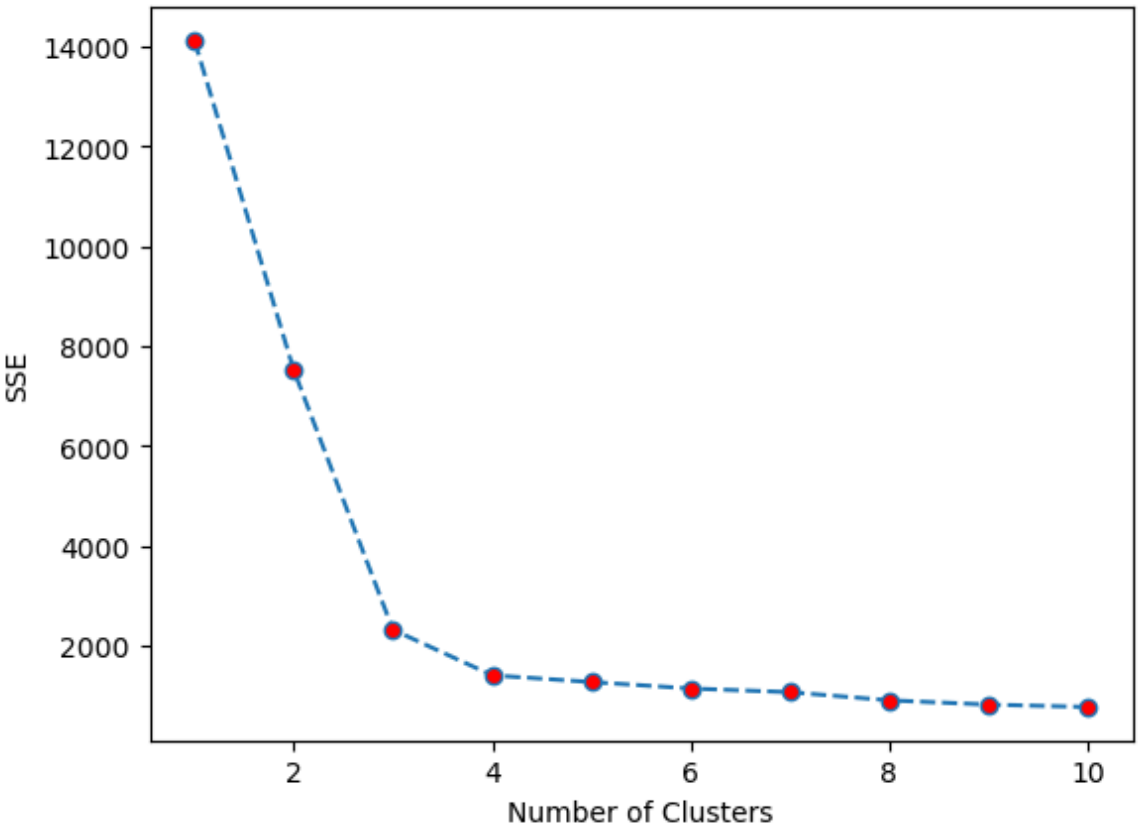
In [20]:
```
sse = []

for i in range(1, 11):
```

```
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(features)
    sse.append(kmeans.inertia_)

plt.plot(range(1, 11), sse, marker='o', markerfacecolor='red', linestyle='

plt.xlabel('Number of Clusters')
plt.ylabel('SSE')
plt.show()
```



And we have a graph of `sse vs k` where we can see that the `sse` for `k=4` is the when it starts to sharply slow down and then it decreases by a small margin and we can select `k=4` as the best `k` for the data.

And that's it for `k means clustering`.

Final Words

Colors and values mean nothing.