```python
import numpy as np
import pandas as pd
```

```python
# @title How our Data-set looks like
dataset = pd.read_csv('Mall_Customers.csv')
dataset.head()
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

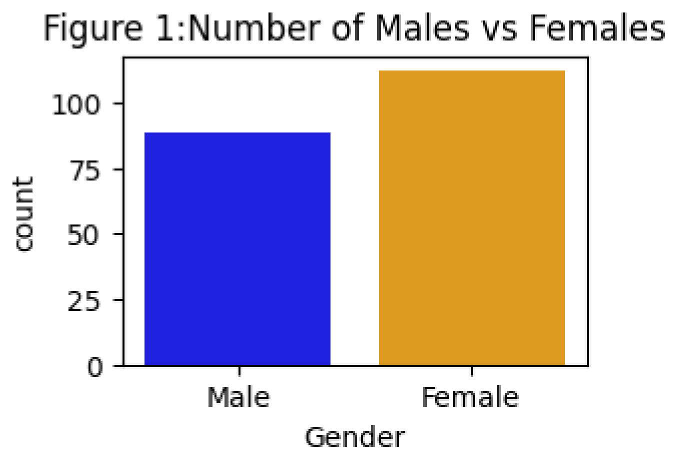
Data-Analysis and Insights

The spending score is the indication of how much a customer spends in our particular mall. This is easily the most valuable information we have, so our analytics will be heavily based around this feature.

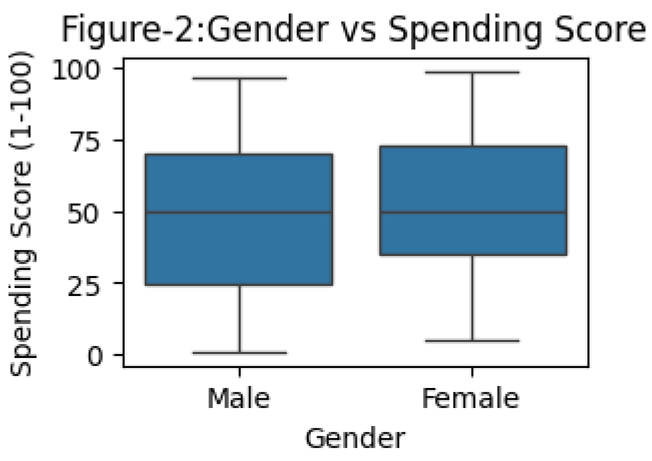
Gender

Here we can see that there are more female cutomers than male customers. Also, the spending score of females is higher than that of males(as seen in the box-plot(figure-2)). Using such information, the mall can prioritize products/services targeted towards female customers. However, in this particular case, the difference in spending score of different genders and the number of customers belonging to each gender group isn't significant. The spending score more or less seems to be independent of the Gender(which can be seen in Figure-3 below). So we may drop this feature while training our model.

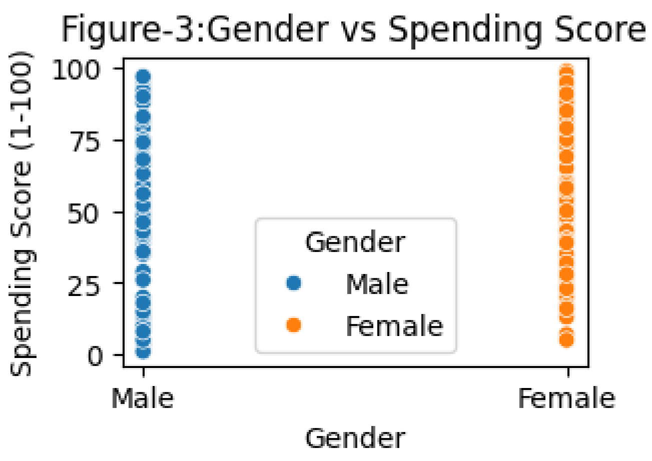
```python
# @title *Customer Division*
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(3,2))
sns.countplot(x='Gender',hue='Gender',data=dataset,palette=['blue','orange'],dodge=False,legend=False)
plt.title('Figure 1:Number of Males vs Females')
plt.show()
```



```python
# @title *Gender and spending score*
plt.figure(figsize=(3,2))
sns.boxplot(x='Gender', y='Spending Score (1-100)', data=dataset)
plt.title('Figure-2:Gender vs Spending Score')
plt.show()
```



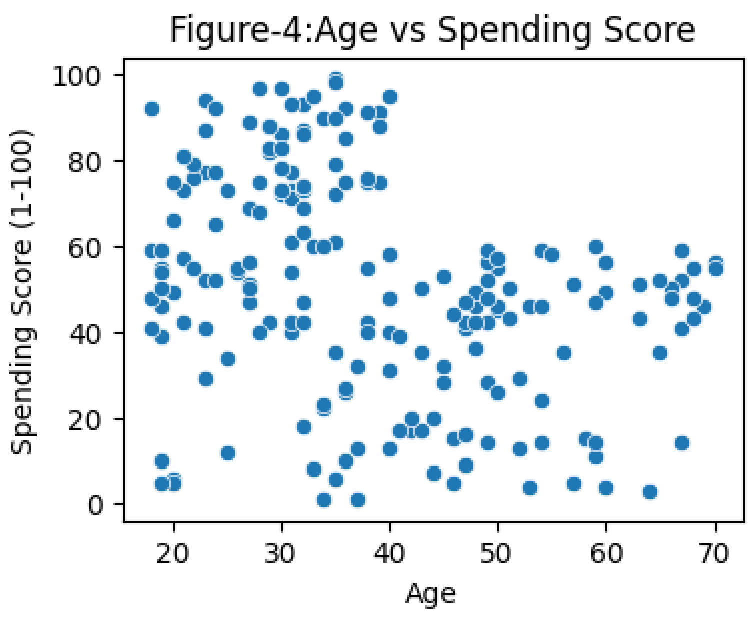
```python
# @title *Spending-score is independent of Gender*
import matplotlib.pyplot as plt
import seaborn as sns

df3 = dataset[['Gender','Spending Score (1-100)']].copy()

plt.figure(figsize=(3,2))
sns.scatterplot(x='Gender', y='Spending Score (1-100)', hue='Gender', data=df3)
plt.title('Figure-3:Gender vs Spending Score')
plt.show()
```



Age

From the scatterplot, we can intuitively observe distinct clusters. Like the cluster at the top left corner of figure-4, which depicts a group of young customers having high spending score. Using this information, the mall can advertise products like high-end wearable or gadgets to this group. There's a good chance that that group of customers would be inclined to buy those products, hence profiting the mall. This was just one example. Already we can see the usefulness of 'Age' feature unlike 'Gender'.

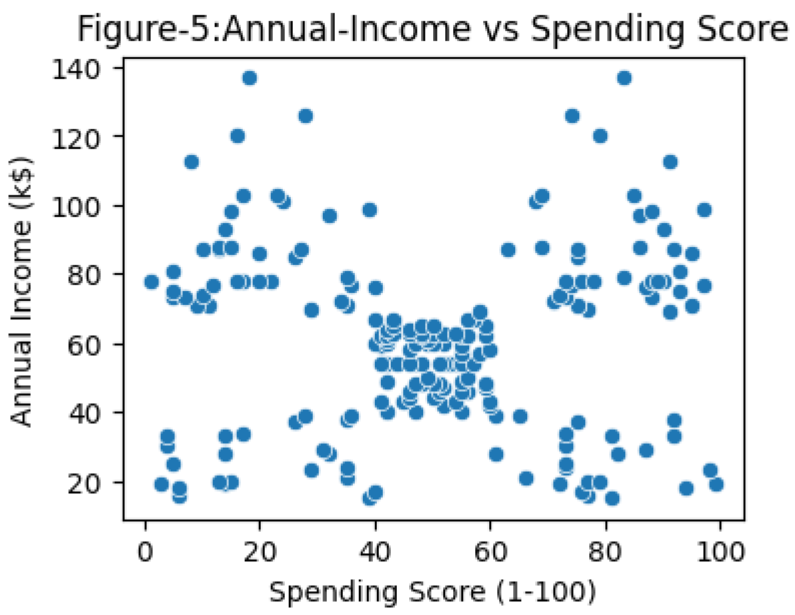
```python
# @title *Age vs Spending-score*
df4 = dataset[['Age','Spending Score (1-100)']].copy()

plt.figure(figsize=(4,3))
sns.scatterplot(x='Age', y='Spending Score (1-100)', data=df4)
plt.title('Figure-4:Age vs Spending Score')
plt.show()
```



Annual-Income

Here we can observe an even clearer distinction of customers than we did for the previous feature. Here we can obtain very useful and actionable insights.Customres with high income and high spending can be targeted with luxury products. Customers with low income and low spending can be attracted via discounts. Standard marketing for customers with average income and average spending, and so on.

```
# @title *Annual-Income vs Spending-score*
df5 = dataset[['Annual Income (k$)','Spending Score (1-100)']].copy()

plt.figure(figsize=(4,3))
sns.scatterplot(y='Annual Income (k$)', x='Spending Score (1-100)', data=df5)
plt.title('Figure-5:Annual-Income vs Spending Score')
plt.show()
```



The K-Means class

Code

```python
import matplotlib.pyplot as plt

class KMeans:
  def __init__(self,X,num_clusters,max_iters):
    self.num_clusters = num_clusters
    self.max_iters = max_iters
    self.num_examples,self.num_features = X.shape #rows = number of examples
                                                  #columns = number of features

  def initialize_centroids(self,X):
    centroids = np.zeros((self.num_clusters,self.num_features))
    for cluster_id in range(self.num_clusters):
      centroid = X[np.random.choice(range(self.num_examples))]
      centroids[cluster_id] = centroid
    return centroids

  def create_clusters(self,centroids,X):
    clusters = [[] for _ in range(self.num_clusters)]
    for point_id,point in enumerate(X):
      closest_centroid = np.argmin(np.sqrt((np.sum((point-centroids)**2,axis=1))))
      clusters[closest_centroid].append(point_id)
    return clusters

  def calculate_new_centroids(self,clusters,X):
    centroids = np.zeros((self.num_clusters,self.num_features))
    for cluster_id,cluster in enumerate(clusters):
      if(len(cluster) == 0):
        centroids[cluster_id] = X[np.random.choice(self.num_examples)]
      else:
        new_centroid = np.mean(X[cluster],axis=0)
        centroids[cluster_id] = new_centroid
    return centroids

  def train(self,X):
    centroids = self.initialize_centroids(X)
    for _ in range(self.max_iters):
      clusters = self.create_clusters(centroids,X)
      previous_centroids = centroids
      centroids = self.calculate_new_centroids(clusters,X)
      diff = centroids - previous_centroids
      if not diff.any():
        break
    return clusters,centroids

  def plot(self, X, clusters, centroids,labels):
      colors = ['red', 'blue', 'green', 'cyan', 'magenta', 'yellow', 'black', 'orange', 'purple', 'brown

      for cluster_id, cluster in enumerate(clusters):
          points = X[cluster]  # Get all points for this cluster
          plt.scatter(points[:, 0], points[:, 1], s=50, c=colors[cluster_id % len(colors)], label=f'Clus

      # Plot centroids
      plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='black', marker='X', label='Centroids')

      plt.title('Clusters of Customers')
      plt.xlabel(labels[0])
      plt.ylabel(labels[1])
      plt.legend()
      plt.show()
```

Working

init

'init' function of a class is automatically called when we create a new instance of that class.

This function assigns values to important parameters(number of clusters,maximum number of iterations etc) of our K-means model.

Initialize Centroids

We have created a 2-d array to hold the initial values of centroids. 'np.zeros((self.num_clusters,self.num_features))' creates an array with (rows = number of clusters) and (columns = number of features we are considering). Say num_clusters = 5 and num_features = 3. This means we have 5 clusters, so we need to have 5 centroids and each centroid will be represented by a vector of dimension three. From the code below we got

[[18. 48. 59.]

[32. 60. 42.]

[33. 113. 8.]

[70. 46. 56.]

[45. 28. 32.]]

This means for centroid 1, the age value is 18,annual income is 48 and spending score is 59. These values are taken from random training examples in our data-set.

Note: When initializing the centroids randomly, we may run into 'Random Initialization Trap' problem which occurs when the centroids are initialized too close to each other. Then we may not get the desired clusters.

This problem is dealth with by an upgraded implementation k/a K-means++.

```
fortest = dataset.iloc[:,[2,3,4]].values
testmodel = KMeans(fortest,5,100)
testcentroids = testmodel.initialize_centroids(fortest)
print(testcentroids)
```

```
[[ 29.   73.   88.]
 [ 23.   16.   77.]
 [ 35.  120.   79.]
 [ 28.   87.   75.]
 [ 50.   40.   55.]]
```

Create Clusters

We initialized 'clusters' as a list of list.

clusters = [cluster1,cluster2,...,cluster_k]

Each individual cluster stores the datapoints that belongs to it.

cluster1 = [datapoint1,datapoint5..etc]

The line we need to understand in this function:

```
closest_centroid = np.argmin(np.sqrt((np.sum((point-centroids)^2,axis=1))))
```

> > **point - centroids**

'point' is a single point like `[2.5, 3.0]` (a 2D point). 'centroids' is a list/array of all centroids, like:

```
[[1.0, 1.0],
 [5.0, 5.0],
 [2.0, 2.5]]
```

When we do 'point-centroids', it subtracts each row in centroids from `[2.5, 3.0]` .

`[[ 1.5,  2.0]` , # (2.5-1.0, 3.0-1.0)

`[-2.5, -2.0]` , # (2.5-5.0, 3.0-5.0)

`[ 0.5,  0.5]]` # (2.5-2.0, 3.0-2.5)

> > **(point-centroids)^2**

squares each entry in `(point-centoids)`

```
[[2.25, 4.00],
 [6.25, 4.00],
 [0.25, 0.25]]
```

> > **np.sum(..., axis=1)** axis=1 means we are taking the sum 'row-wise',

First centroid: 2.25 + 4.00 = 6.25

Second centroid: 6.25 + 4.00 = 10.25

Third centroid: 0.25 + 0.25 = 0.5

which gives us `[6.25, 10.25, 0.5]`

So essentially, till this point we have calculated the sum of squared distances between the point and each centroid. To understand it better, say if the point is (a1,a2) and n-th centroid c_n is (b1,b2), then we have calculated (a1-b1)^2 + (a2-b2)^2 (which is exactly what we need!). This works for any number of features. Say if we took 'k' features then the point could be represented as (a1,a2,..,a_k) and arbitrary cluster c_n = (b1,b2,...,b_k), then the output would be (a1-b1)^2 + ... + (a_k-b_k)^2.

> > **np.sqrt(...)**

This step is not important for the working of K-means algorithm Square-root is a monotonically increasing function, so if 'a' is the smallest out of (a,b,c,d) then sqrt(a) will be smallest out of sqrt(a,b,c,d).

But for the sake of the distance being the actual Euclidean-distance, we are taking the square-root.

`[2.5, 3.2, 0.707]`

> > **np.argmin(...)**

This gives the index of the smallest value. 0.707 is the smallest, so it return 2. Hence the point is assigned to cluster-3 via `clusters[closest_centroid].append(point_id)`

This is done for each data-point in the dataset.

Calculate new centroids

A centroid is like a representative to its cluster, so it should reflect its properties. The randomly initialized centroids obviously won't do that. We redefine each centroid by taking the mean of all the data-points assigned to that centroid.

```
new_centroid = np.mean(X[cluster],axis=0)
```

> > **X[cluster]**

X contains all the data points from our dataset so we extract only the points that are assigned to cluster number-cluster_id. `X = [[1.0, 2.0],  [2.0, 3.0],  [4.0, 5.0],  [12.0,6.0],  ... ]` `X[cluster] = [[1.0, 2.0],  [4.0, 5.0],  [7.0, 8.0]]`

> > **np.mean(..,axis=0)**

Each cluster may hold multiple datapoints, say

```
X[cluster] =
[[1.0, 2.0],
 [4.0, 5.0],
 [7.0, 8.0]]
```

axis=0 means we are taking mean column-wise, i.e. individually for each feature(which is what we want to calculate mean).

(1.0 + 4.0 + 7.0) / 3 = 12.0 / 3 = 4.0

(2.0 + 5.0 + 8.0) / 3 = 15.0 / 3 = 5.0

Thus,

```
new_centroid = [4.0, 5.0]
```

Train

All the hard work has been done! Now we just need to use the above mentioned function to train the model.

1. Initialize the centroids.

Repeat: 2. Create clusters. 3. Find the new centroids using these clusters 2. Find the difference between current and previous centroids

Until:

1. maximum iterations have been reached. Or,
2. current centroids = previous centroids.

When this last condition is reached, then we say that the model has converged. The centroids are fixed and beyond this point they won't change for any number of iterations.

Technically convergence is reached when no centroid is changed && no point has been assigned a different cluster, betweeen two consecutive iterations. But that is costly to track and only tracking the centroids will suffice for our purposes.

How many clusters should we take?

We are using the 'Elbow-method' to find the optimal number of clusters.

We define the term WCSS(Within-Cluster Sum of Squares) as the sum of squares of distances between all the points and their corresponding centroids.

It measures how tightly the data points are clustered around their centroids. We can use this as a metric of quality. One thing to note is that wcss always decreases as the number of clusters increase. But if we keep increasing the number of clusters, we eventually overfit — we're just reducing WCSS without meaningful grouping. So we choose the point where the improvement in wcss drops significantly, that point is known as the 'elbow-point'.

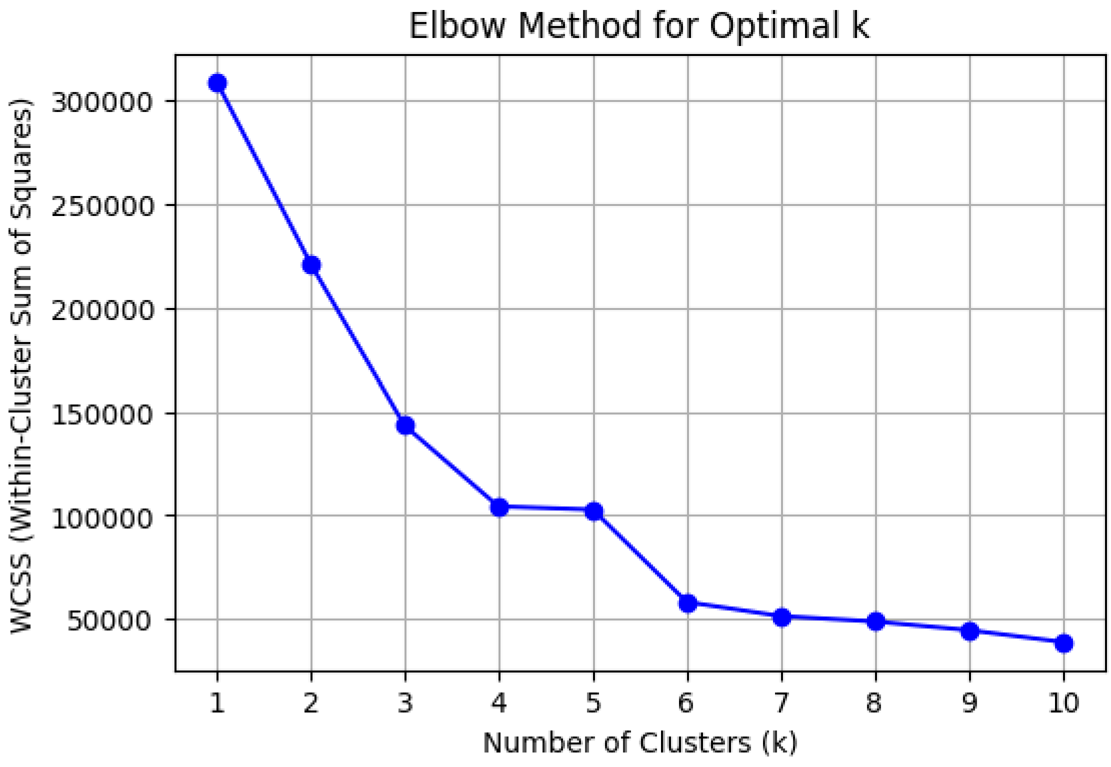
Code

```
def find_wcss(X,clusters,centroids):
  wcss = 0
  for cluster_id,cluster in enumerate(clusters):
    for point_id in cluster:
      diff = X[point_id] - centroids[cluster_id]
      distance = np.sqrt(np.sum(diff ** 2))
      wcss += distance ** 2
  return wcss

def elbow_method(X, max_k=10, max_iters=100):
    wcss = []

    for k in range(1, max_k + 1):
        model = KMeans(X, num_clusters=k, max_iters=max_iters)
        clusters, centroids = model.train(X)
        wcss.append(find_wcss(X,clusters,centroids))

    # Plotting the elbow curve
    plt.figure(figsize=(6, 4))
    plt.plot(range(1, max_k + 1), wcss, 'bo-')
    plt.xlabel('Number of Clusters (k)')
    plt.ylabel('WCSS (Within-Cluster Sum of Squares)')
    plt.title('Elbow Method for Optimal k')
    plt.xticks(range(1, max_k + 1))
    plt.grid(True)
    plt.show()
```

Plot

Let's plot the elbow-curve to better understand this method. For this demonstration, we're using the features 'Age','Annual Income' and 'Spending Score'.

```
X = dataset.iloc[:,[2,3,4]].values
elbow_method(X)
```



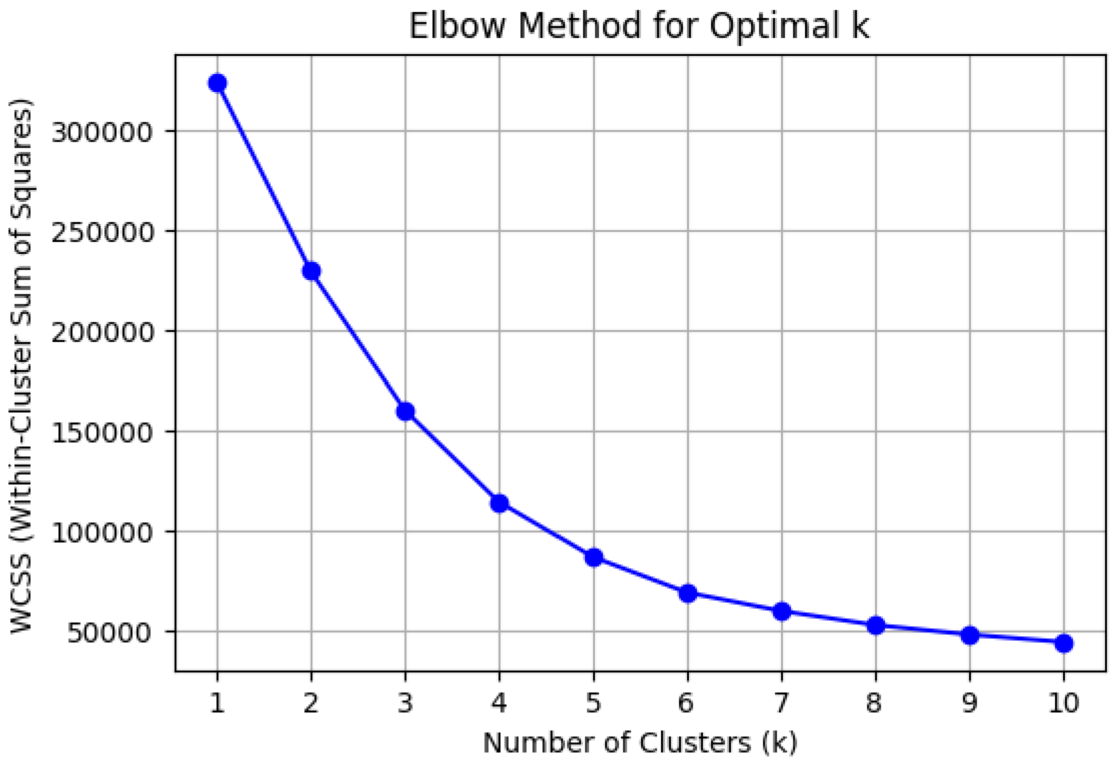
Here if you notice, the wcss isn't always decreasing with increasing number of clusters. We're getting random spikes in the curve. This happens because of Random Initialization Noise. Since we pick centroids completely at random, some runs will just be “unlucky” and produce a very poor clustering, spiking the sum for that single run. To counter the unlucky runs, we will run the model multiple times for each k and find the average wcss value.

Improved Code

```python
def elbow_method_improved(X, max_k=10, max_iters=100):
    wcss = []

    for k in range(1, max_k + 1):
      sum = 0
      for runs in range(21):
        model = KMeans(X, num_clusters=k, max_iters=max_iters)
        clusters, centroids = model.train(X)
        sum += find_wcss(X,clusters,centroids)
      wcss.append(sum/20)

    # Plotting the elbow curve
    plt.figure(figsize=(6, 4))
    plt.plot(range(1, max_k + 1), wcss, 'bo-')
    plt.xlabel('Number of Clusters (k)')
    plt.ylabel('WCSS (Within-Cluster Sum of Squares)')
    plt.title('Elbow Method for Optimal k')
    plt.xticks(range(1, max_k + 1))
    plt.grid(True)
    plt.show()
```

Improved Plot

```python
elbow_method_improved(X)
```



Which features should we include?

We will find the wcss-score for various feature combinations and decide from there.

We will fix the number of clusters= 5, which weobtained from above. And again, we will run themodel multiple times and take the average.

Age-Income-Spending

```
X_a = dataset.iloc[:,[2,3,4]].values

wcss_a = 0
for i in range(101):
  model_a = KMeans(X_a,5,100)
  clusters_a,centroids_a = model_a.train(X_a)
  wcss_a += find_wcss(X_a,clusters_a,centroids_a)
wcss_a /= 100
print(wcss_a)
```

```
86425.18422988543
```

Income-Spending

```
X_b = dataset.iloc[:,[3,4]].values

wcss_b = 0
for i in range(101):
  model_b = KMeans(X_b,5,100)
  clusters_b,centroids_b = model_b.train(X_b)
  wcss_b += find_wcss(X_b,clusters_b,centroids_b)
wcss_b /= 100
print(wcss_b)
```

```
51882.13753091169
```

There's a significant reduction in wcss-score after dropping the 'Age' feature, suggesting that we shouldn't use 'Age'. But we saw a very important use-case for it in the Data-Analysis section. There we took Age and Spending-score, let's see that again.

Age-Spending

```
X_c = dataset.iloc[:,[2,4]].values

wcss_c = 0
for i in range(101):
  model_c = KMeans(X_c,5,100)
  clusters_c,centroids_c = model_c.train(X_c)
  wcss_c += find_wcss(X_c,clusters_c,centroids_c)
wcss_c /= 100
print(wcss_c)
```

```
24248.1939307163
```

This gives a good wcss-score, reaffirming our assumption regarding the feature's usefulness.

Age-Income?

This information may be useful for purposes like population census :) but for our purpose, it is not.

Building the final clusters

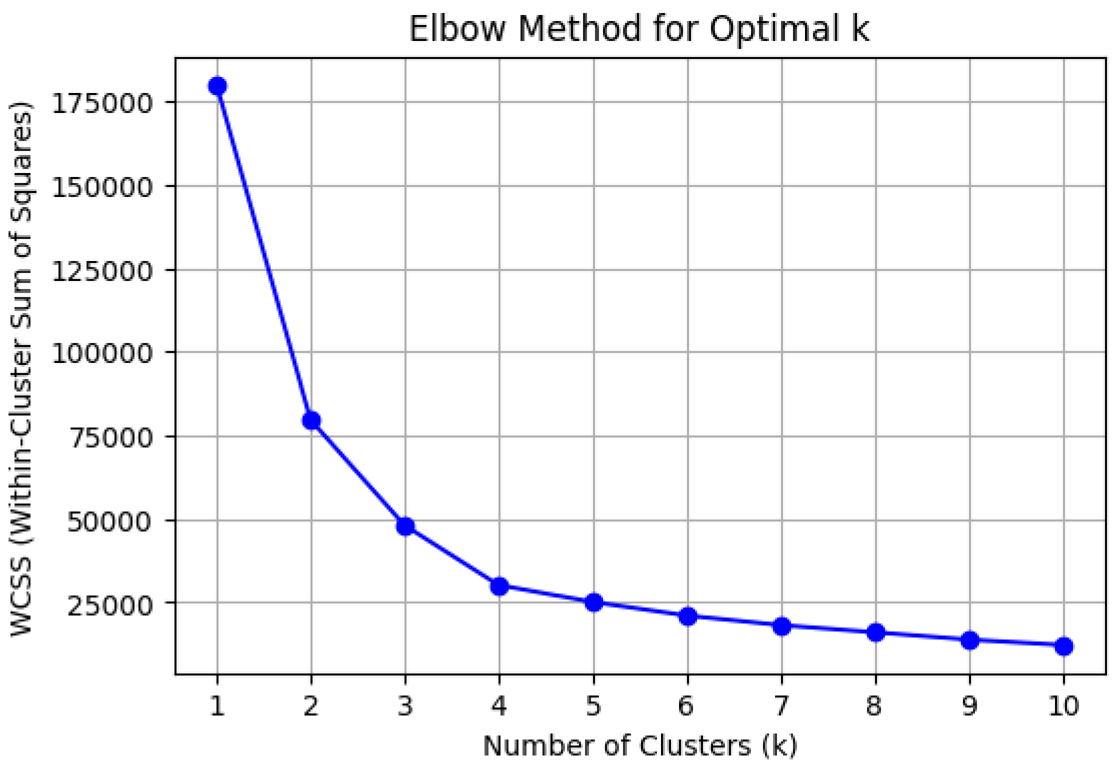
1) using Age & Spending Score

number of clusters?

```
df1 = dataset[['Age','Spending Score (1-100)']].copy()
df1.head()
```

	Age	Spending Score (1-100)
0	19	39
1	21	81
2	20	6
3	23	77
4	31	40

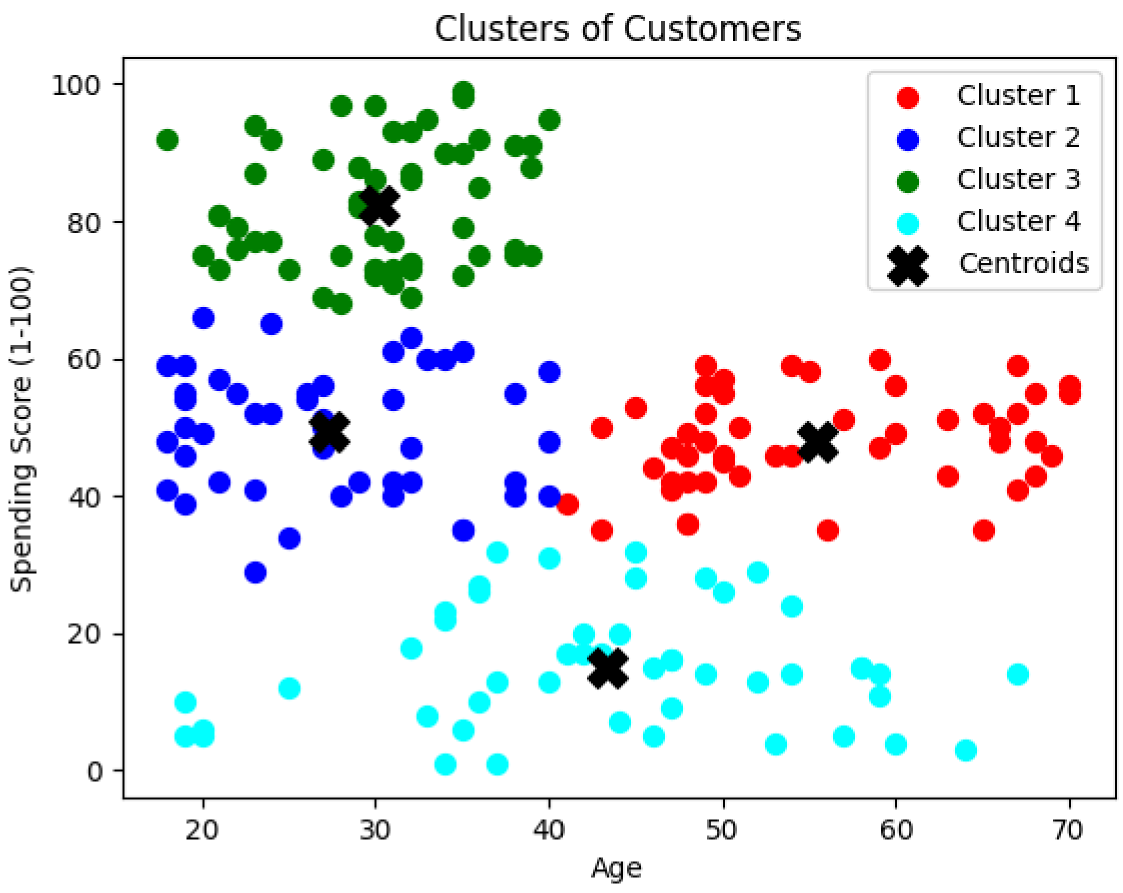
```
X_1 = df1.iloc[:,[0,1]].values
elbow_method_improved(X_1)
```



The wcss-score is decreasing rapidly from k=1 to k=4, from k=4 onwards the score decreases but much slower than before. So we choose num_clusters = 4.

Training the model

```
model1 = KMeans(X_1,4,100)
clusters1,centroids1 = model1.train(X_1)
model1.plot(X_1,clusters1,centroids1,df1.columns)
```



Describing the Groups

Cluster 1: Older, Average Spenders (Red)

- **Description:** Customers in their mid-life to senior years with moderate spending.
- **Strategies:**
- Promote practical products (like health and wellness,travel packages etc).
- Introduce a special membership providing in-home delivery, high-priority customer service, personalized shopping guidance for this group etc.

Cluster 2: Younger, Average Spenders (Blue)

- **Description:** Young adults spending moderately.
- **Strategies:**
- Focus on trendy, affordable products (tech gadgets, fashion, entertainment).
- Social media marketing (paid ads,geo-targeting) can reach them effectively.
- Student Discounts.

Cluster 3: Young, High Spenders (Green)

- **Description:** Young customers willing to spend heavily.
- **Strategies:**
- Target luxury, trendy, premium brands (high-end fashion, technology etc).
- Offer early access to limited editions, VIP events, and premium memberships.
- Aggressively market through influencers and exclusive content campaigns

Cluster 4: Low Spenders Across Ages (Cyan)

- **Description:** Customers with low spending habits, spread across different age groups.
- **Strategies:**
- Focus on low-cost, high-value deals.
- Use email coupons, special limited-time offers, and bundle pricing to boost sales.

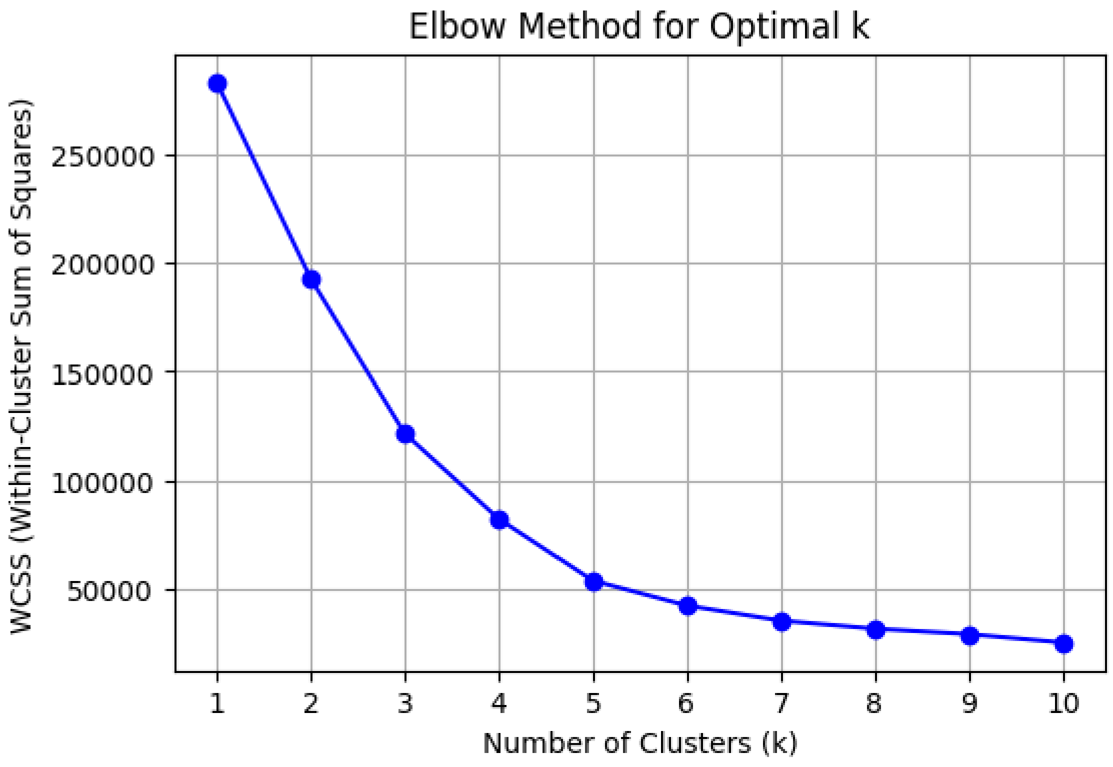
2)using Annual-Income & Spending Score

number of clusters?

```
df2 = dataset[['Annual Income (k$)','Spending Score (1-100)']].copy()
df2.head()
```

	Annual Income (k$)	Spending Score (1-100)
0	15	39
1	15	81
2	16	6
3	16	77
4	17	40

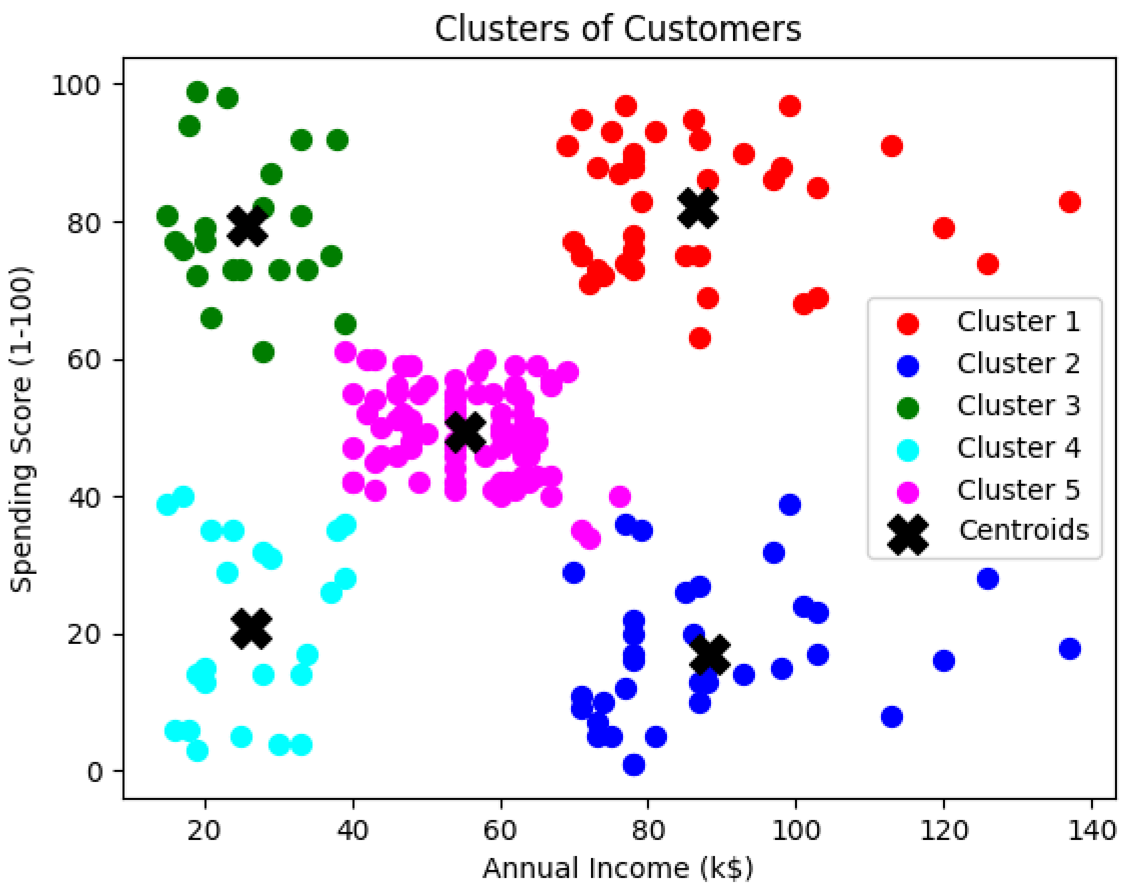
```
X_2 = df2.iloc[:,[0,1]].values
elbow_method_improved(X_2)
```



The elbow clearly appears at k=5.

Training the model

```
model2 = KMeans(X_2,5,100)
clusters2,centroids2 = model2.train(X_2)
model2.plot(X_2,clusters2,centroids2,df2.columns)
```



Describing the Groups

Cluster 1: High Income, High Spending (Red)

- **Description:** Wealthy customers who are willing to spend a lot.
- **Strategies:**
- Focus on luxury shopping experiences (high-end brands, premium stores, gourmet food courts).
- Organize small invitation-only sales previews for new collections before they are opened to the public.
- Hold exclusive giveaways like "Spend above ₹50,000 and win a weekend stay at a luxury hotel".

Cluster 2: High Income, Low Spending (Blue)

- **Description:** Wealthy but cautious spenders.
- **Strategies:**
- Highlight investment-worthy products (quality over quantity: jewelry, luxury electronics).
- Educate through marketing — show how buying at the mall gives them better value or unique products they can't get elsewhere.

Cluster 3: Low Income, High Spending (Green)

- **Description:** Shoppers with lower income but high spending.
- **Strategies:**
- Focus on trendy, affordable stores (fast fashion, tech gadgets, food courts).
- Incentivize repeat visits with mall loyalty cards offering discounts or freebies.

- Influencer tie-ups showcasing affordable shopping deals from the mall.

Cluster 4: Low Income, Low Spending (Cyan)

- **Description:** Price-sensitive group with limited shopping habits.
- **Strategies:**
- Advertise clearance sales, combo offers, and festival discounts.
- Group basic necessity items or frequently bought products together at a discount.
- Offer free trials or samples to encourage product discovery.

Cluster 5: Middle Income, Average Spending (Magenta)

- **Description:** Stable income, middle-class individuals.
- **Strategies:**
- Promote family-centric offers (family movie tickets + food combos).
- Offer mid-range products that are better than budget options but not premium-priced.