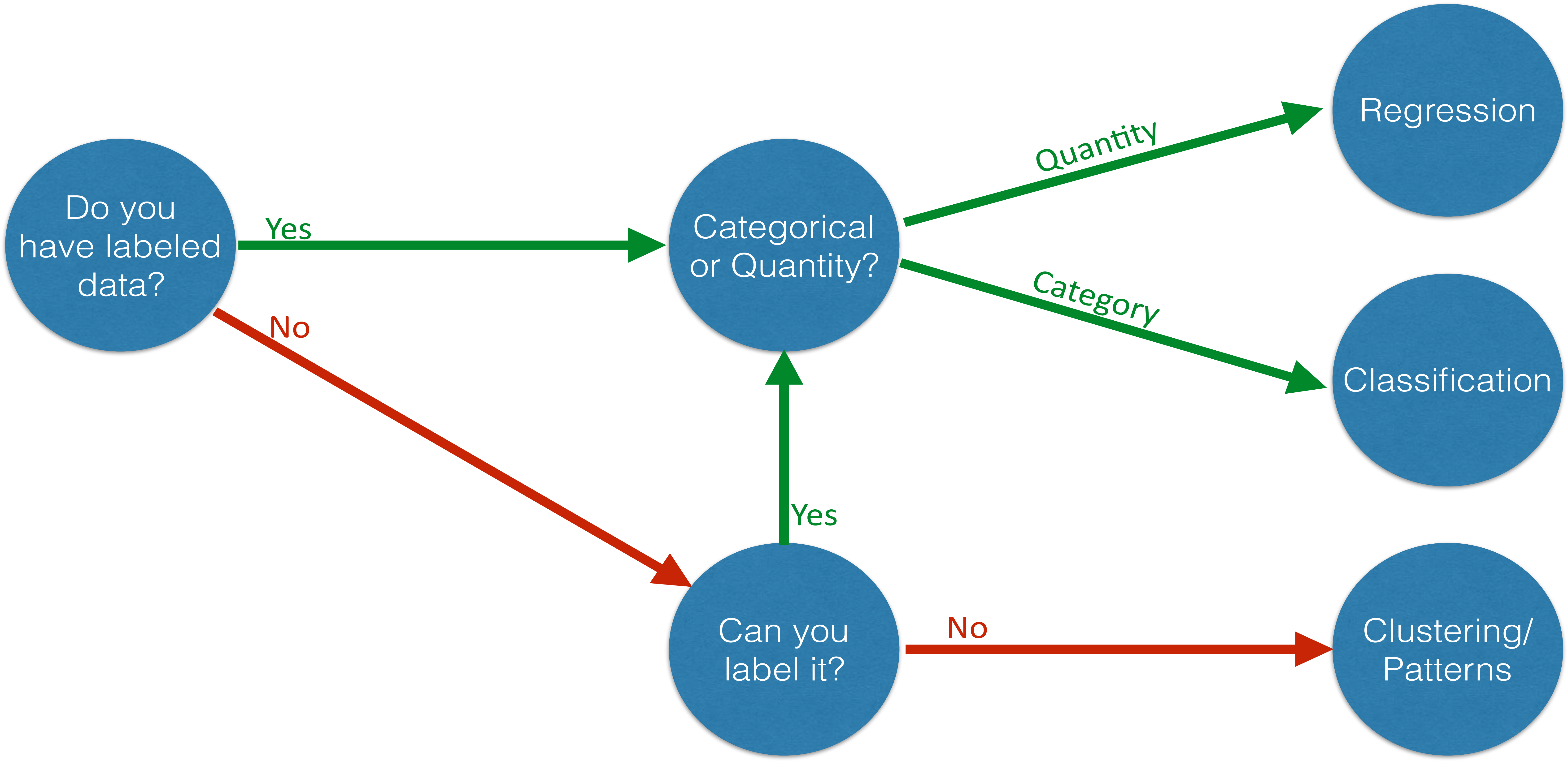
# Module 6: Unsupervised Learning: Clustering

# Agenda for Today

- Measuring Distances
- Math free overview of clustering techniques



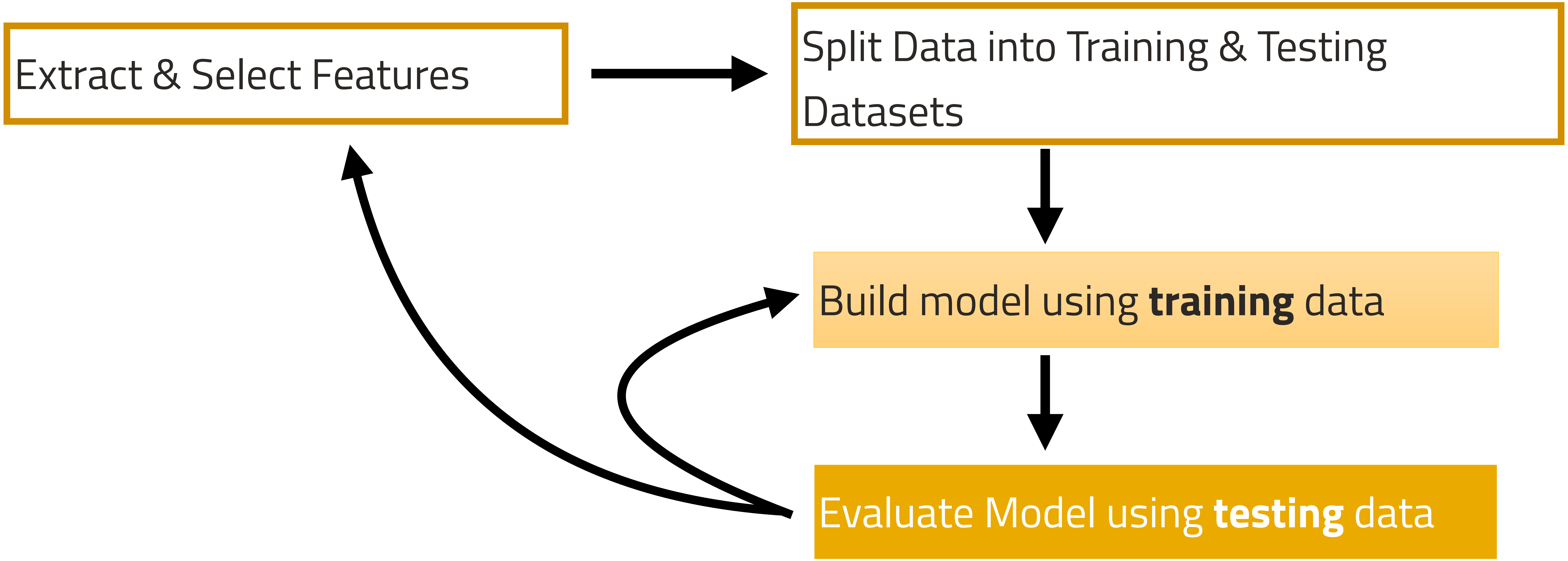

Prediction, Inference

Prediction

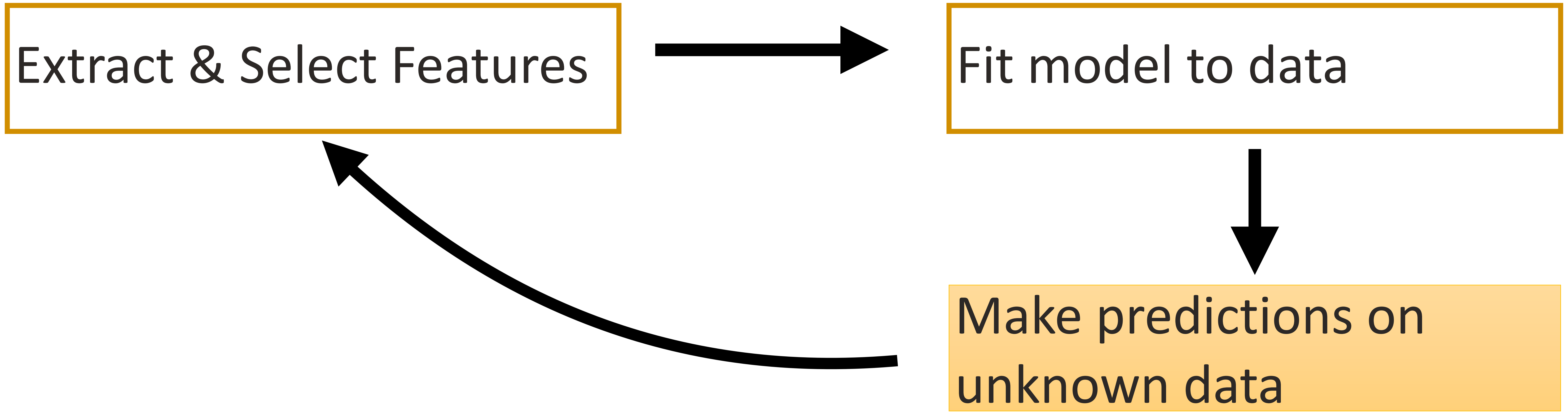
Nothing proven

Recommenders, Similar, anomalous

# Supervised ML Process



# Unsupervised ML Process



# Unsupervised Clustering Algorithm

1. Select Features
2. Calculate a distance measure
3. Apply a clustering algorithm
4. Validate?

# Which Departments are Similar?

| | Malware events |
|---|---|
| Dept1 | 6 |
| Dept2 | 1 |
| Dept3 | 8 |

# Which Departments are Similar?

| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |

# Which Departments are Similar?

| | Malware events | Phishing | Open Tickets |
|---|---|---|---|
| Dept1 | 6 | 6 | 3 |
| Dept2 | 1 | 2 | 1 |
| Dept3 | 8 | 1 | 9 |

# Computing Distance

| | Malware events |
|---|---|
| Dept1 | 6 |
| Dept2 | 1 |
| Dept3 | 8 |

Compare:
Dept1 to Dept2: $|6 - 1| = 5$
Dept2 to Dept3: $|1 - 8| = 7$
Dept1 to Dept3: $|6 - 8| = 2$

# Two-Dimensional Distance

| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |

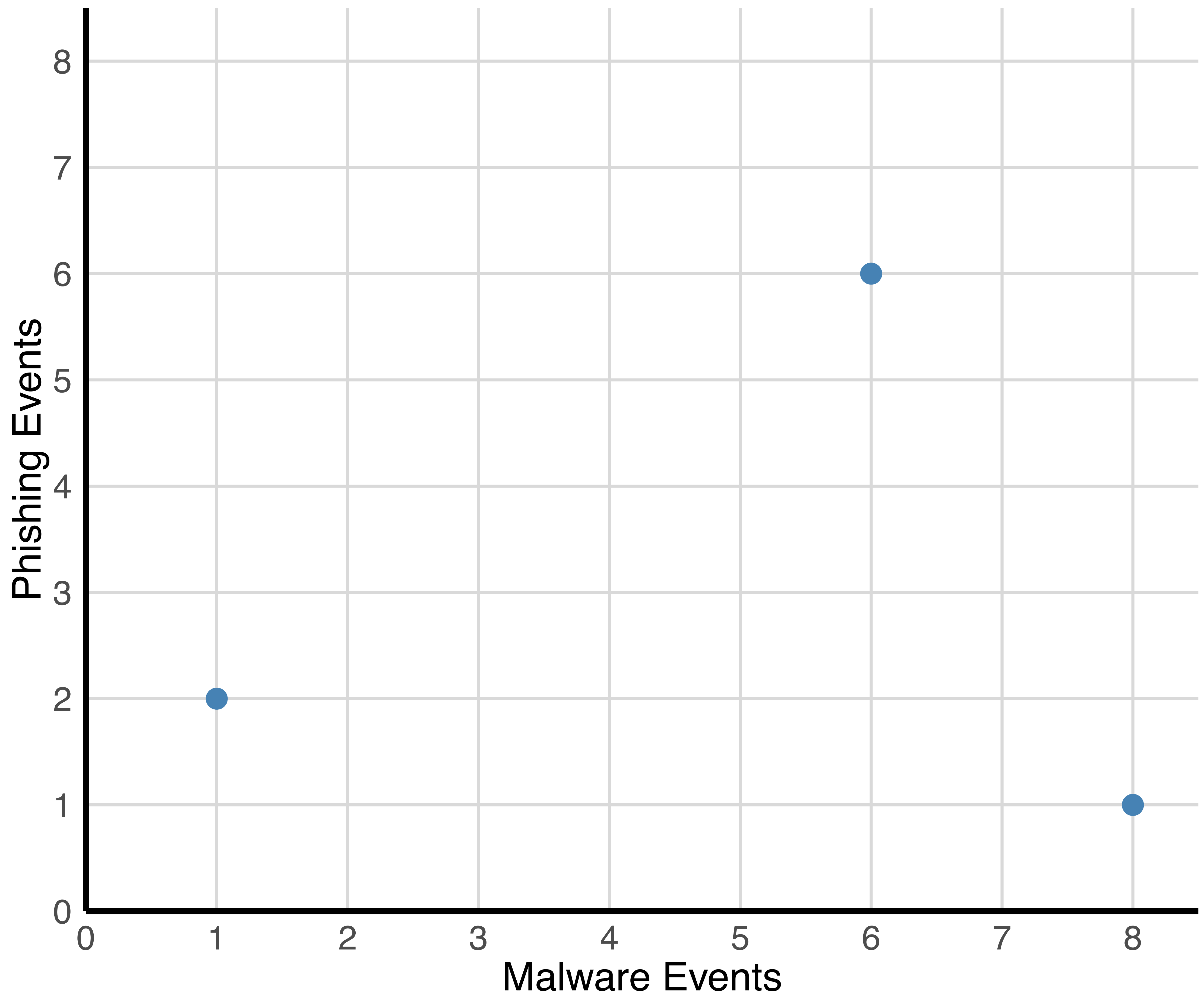
**Multiple Distance methods**

- Euclidean
- Manhattan
- Maximum
- Canberra
- Binary
- Minkowski

… (to name a few)

# Two-Dimensional Distance

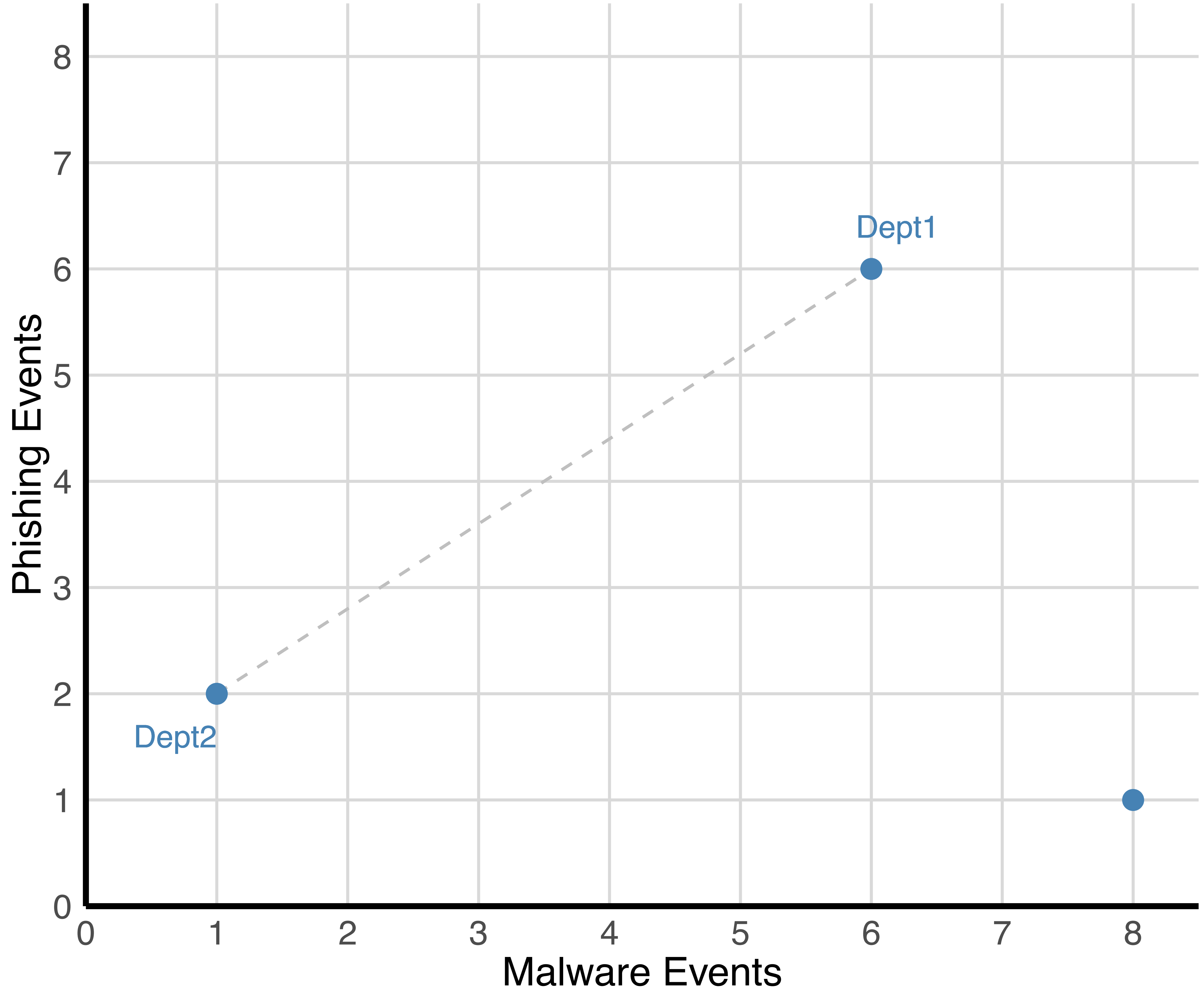
| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |



# Two-Dimensional Distance

## Euclidean very common and easy to understand

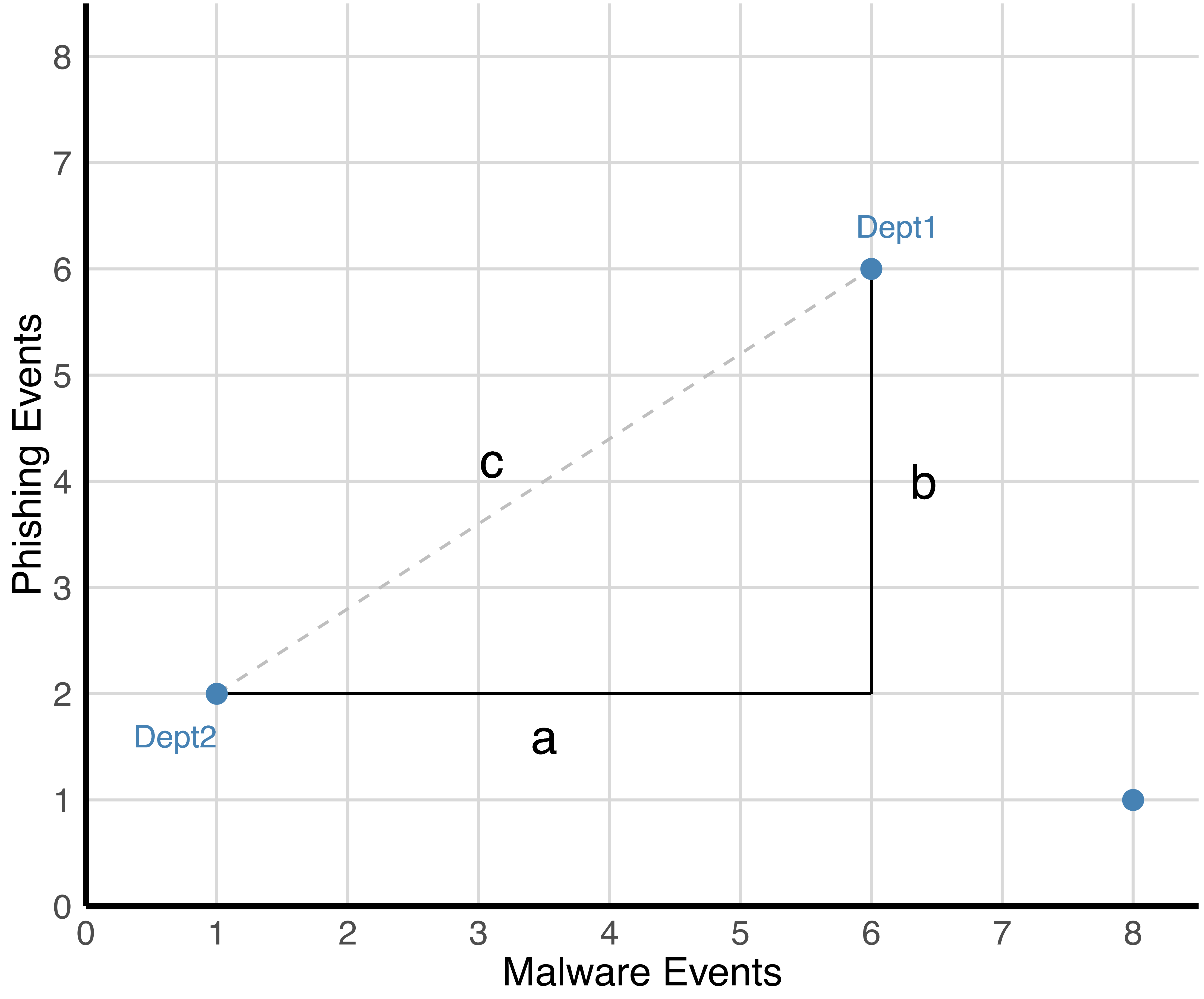
| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |



# Two-Dimensional Distance

Euclidean very common and easy to understand: $a^2 + b^2 = c^2$

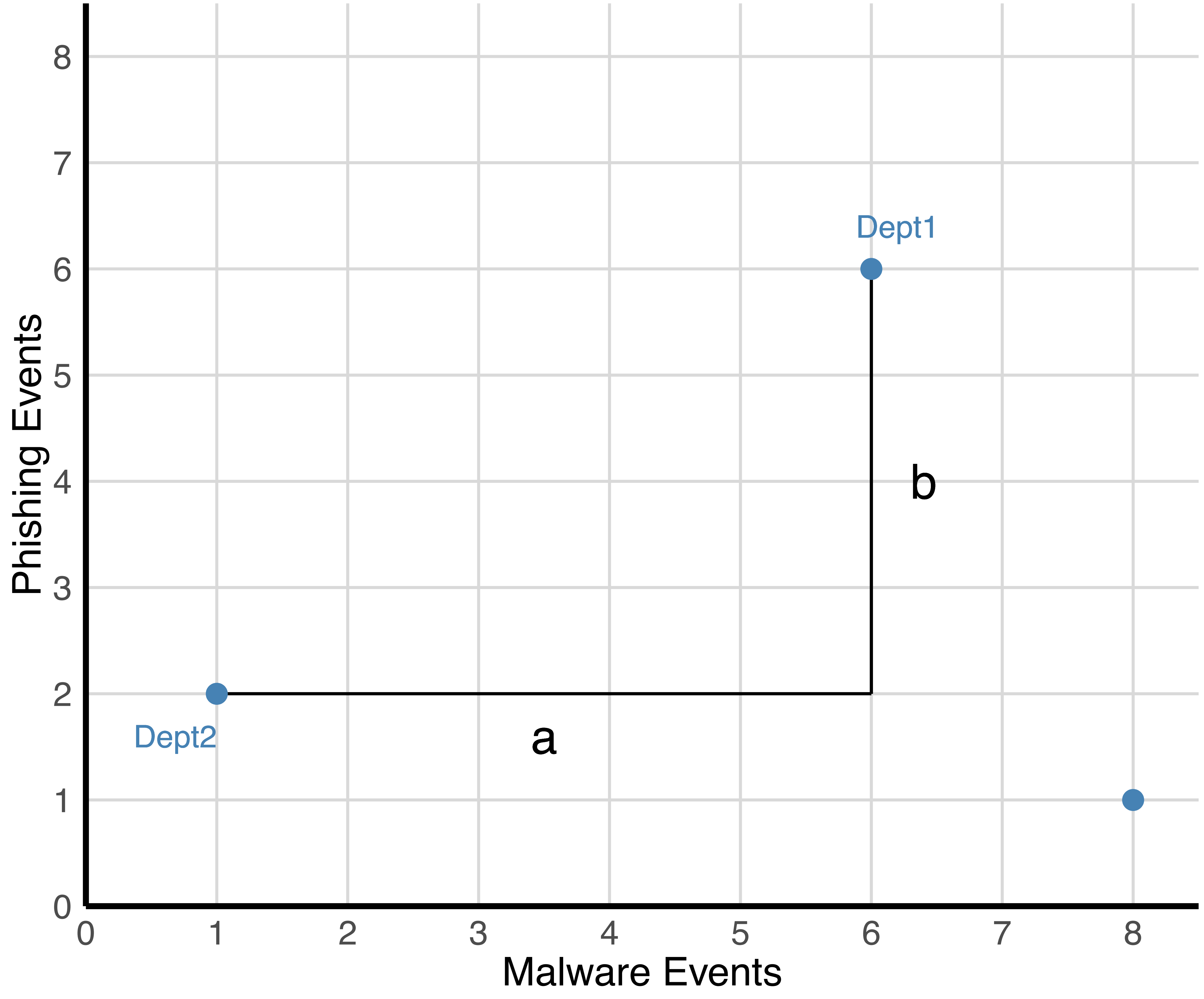
| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |



# Two-Dimensional Distance

Manhattan also easy to comprehend: a + b

| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |



# Computing Distance

| | Malware events | Phishing |
|---|---|---|
| Dept1 | 6 | 6 |
| Dept2 | 1 | 2 |
| Dept3 | 8 | 1 |

Compare:
Dept1 to Dept2: sqrt((6-1)^2 + (6-2)^2) = **6.4**
Dept2 to Dept3: … = **7.1**
Dept1 to Dept3: … = **5.4**

# Euclidean Distance calculations

```
def dist(x,y):
    return np.sqrt(np.sum((x-y)**2))

> mat = np.array([[ 6,6,3 ], [1,2,1], [8,1,9]])
> dist(mat[0], mat[1])
6.7082039324993694

> dist(mat[1], mat[2])
10.677078252031311

> dist(mat[0], mat[2])
8.0622577482985491
```

# Which Departments are Similar?

| | Malware events | Phishing | Open Tickets |
|---|---|---|---|
| Dept1 | 6 | 6 | 3 |
| Dept2 | 1 | 2 | 1 |
| Dept3 | 8 | 1 | 9 |

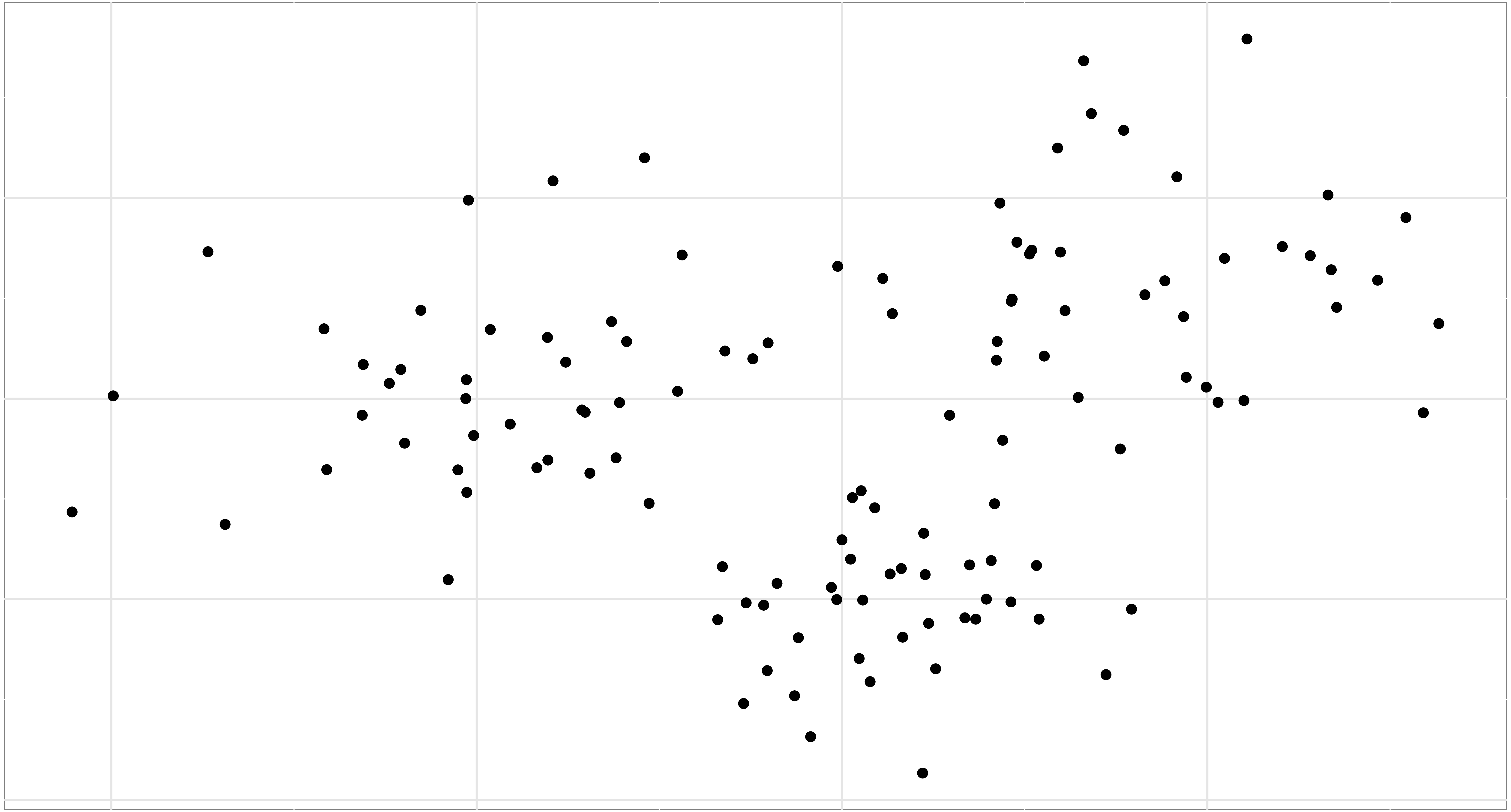
6.7

10.7

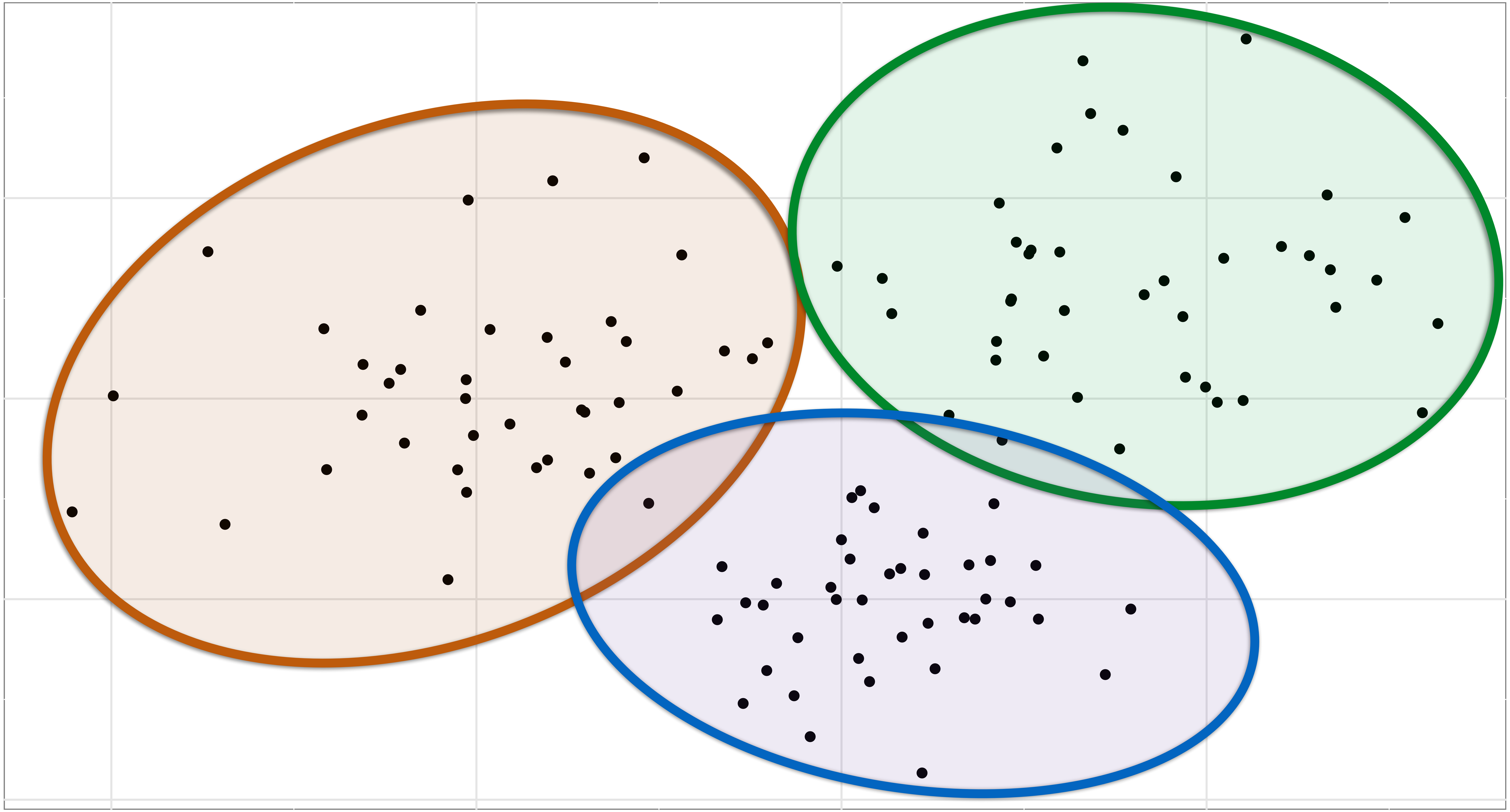
8.1

# Stop

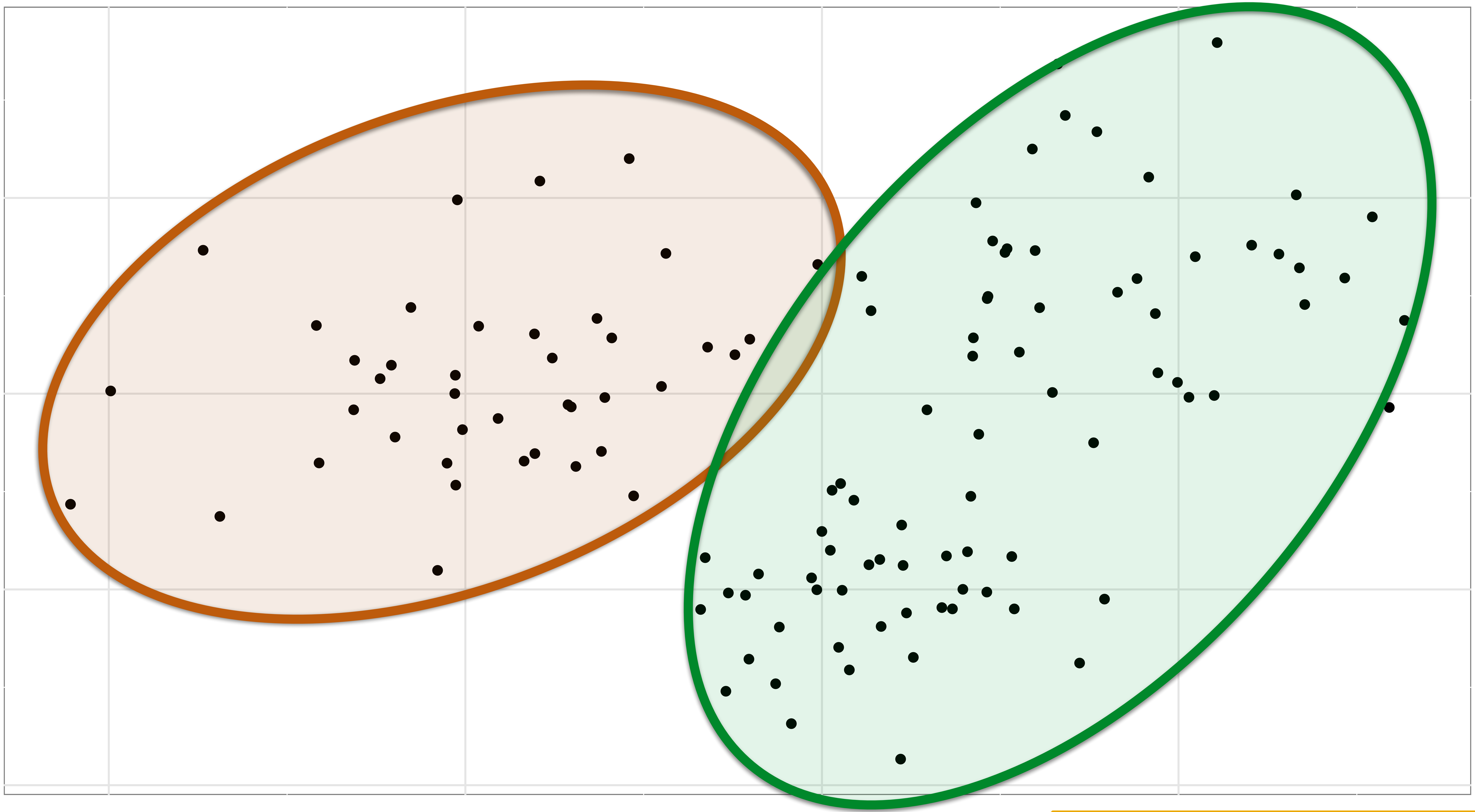
# Clustering…



# Clustering...



# Clustering...

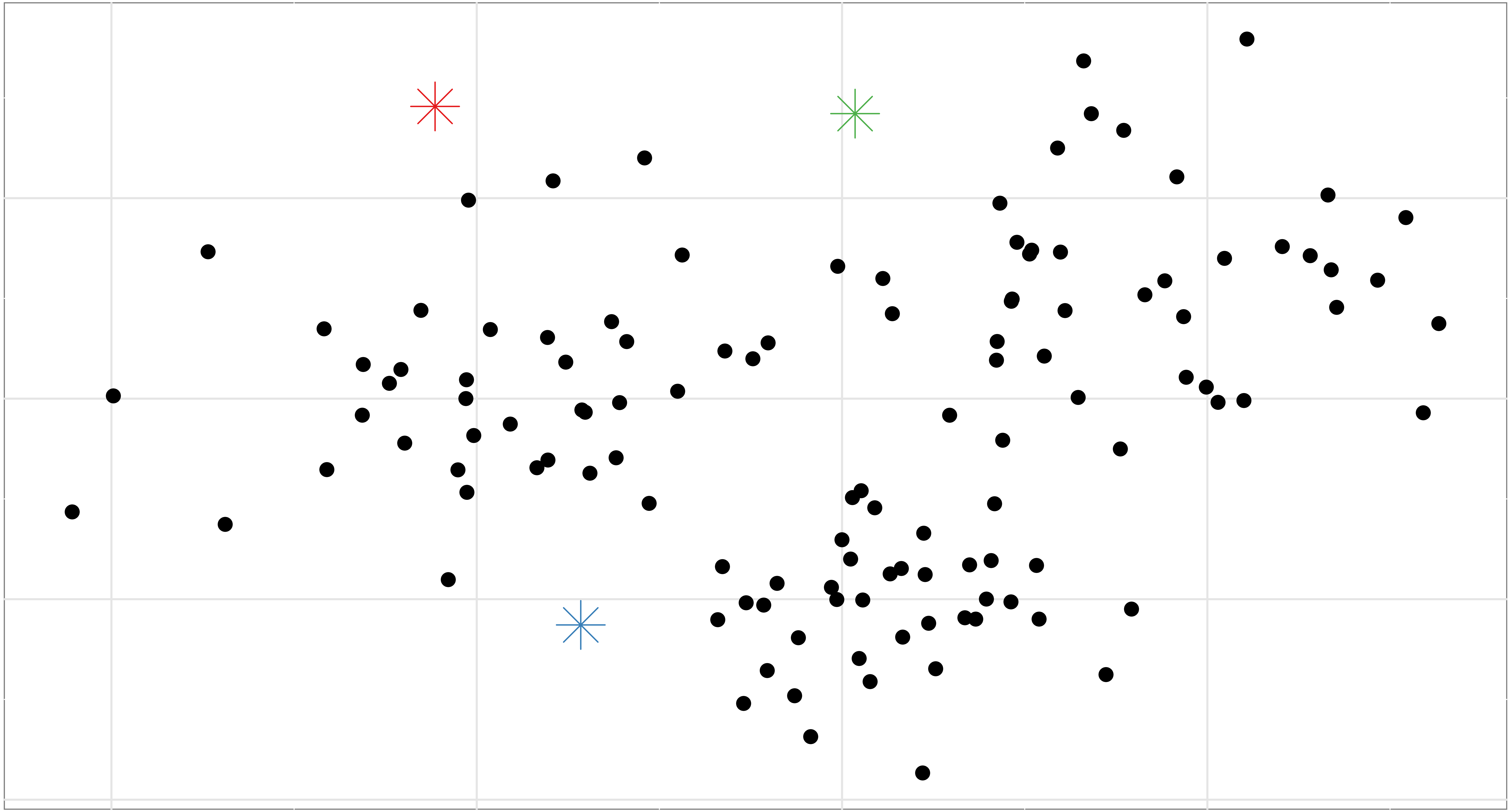


# K-Means

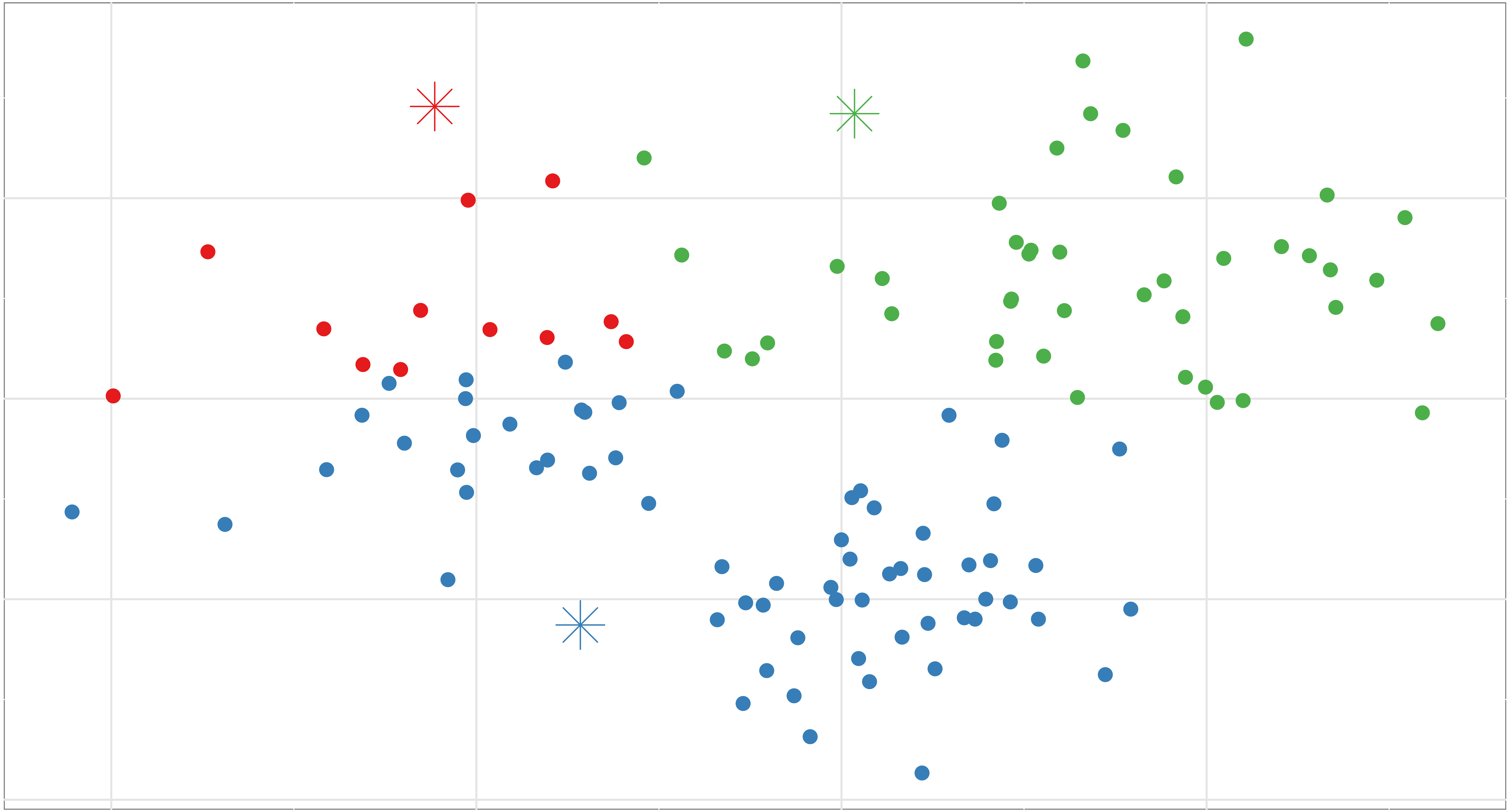
Before starting, pick the number of clusters, K

1. Pick K random centroids within data range
2. Assign each data point to the nearest centroid
3. Move centroid to center of assigned points
4. Repeat steps 2 and 3 until centroid stops shifting

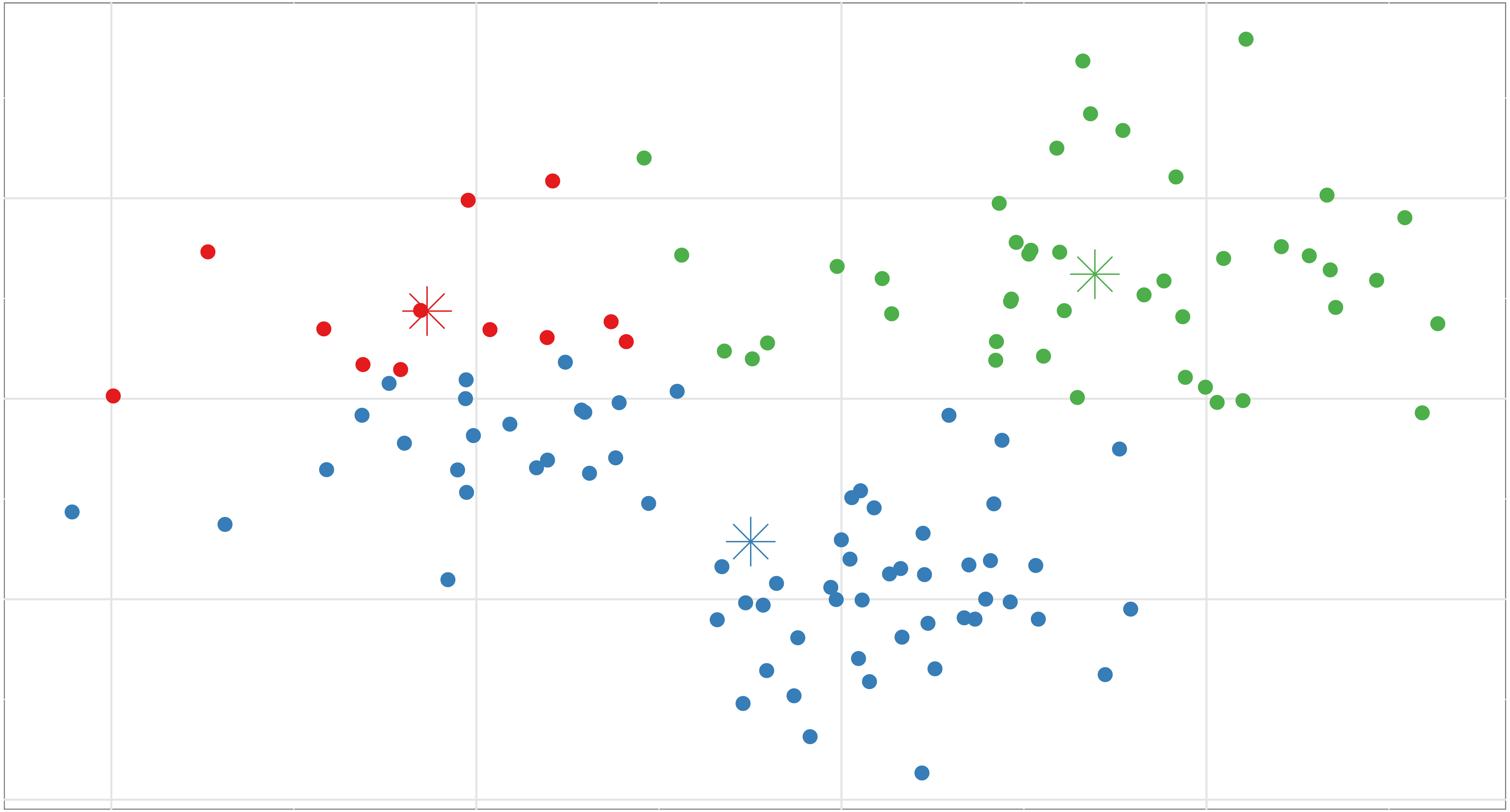
# Step 1: Pick 3 random centroids within data range



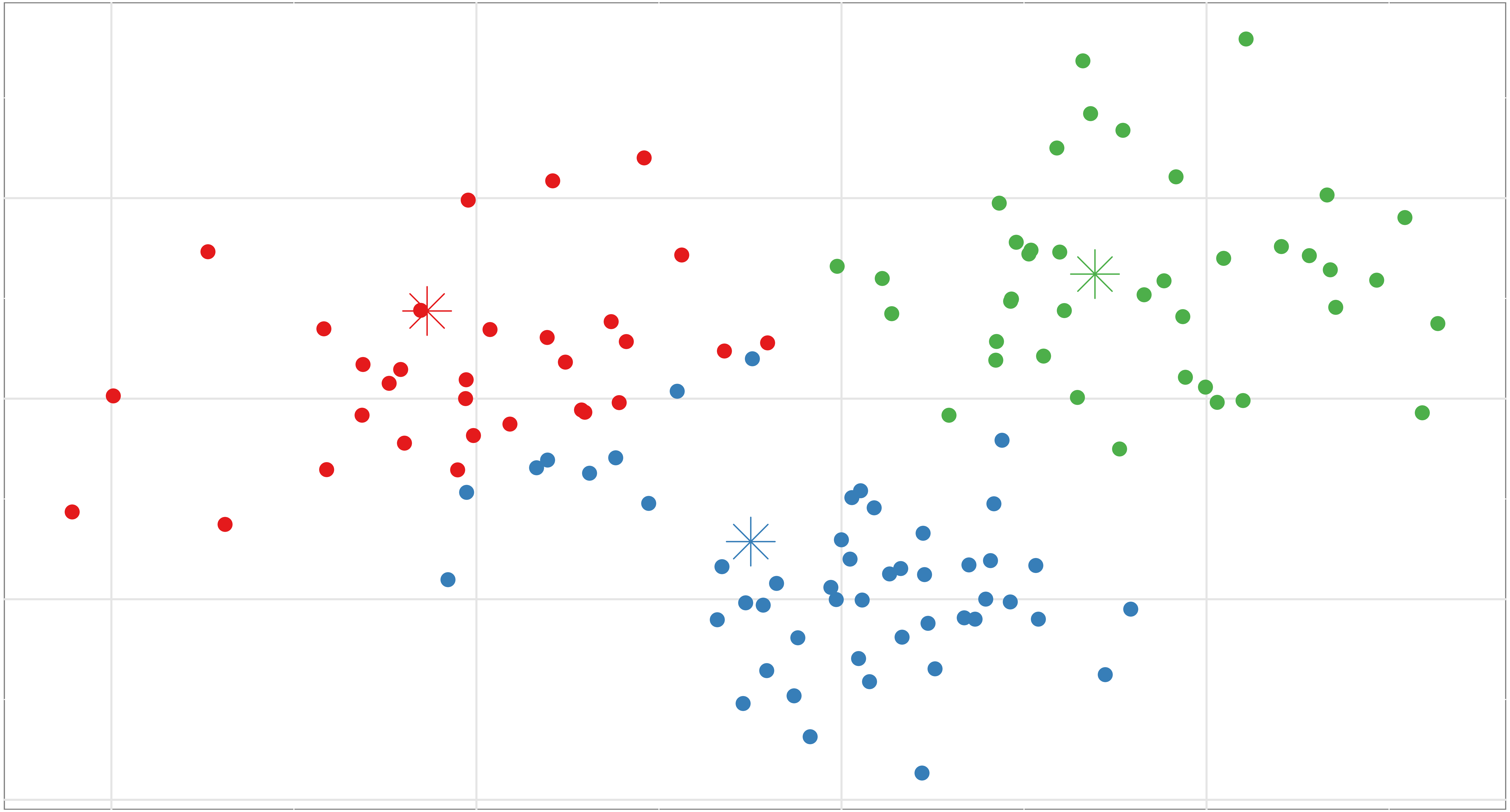
# Step 2: Assign each data point to the nearest centroid (1)



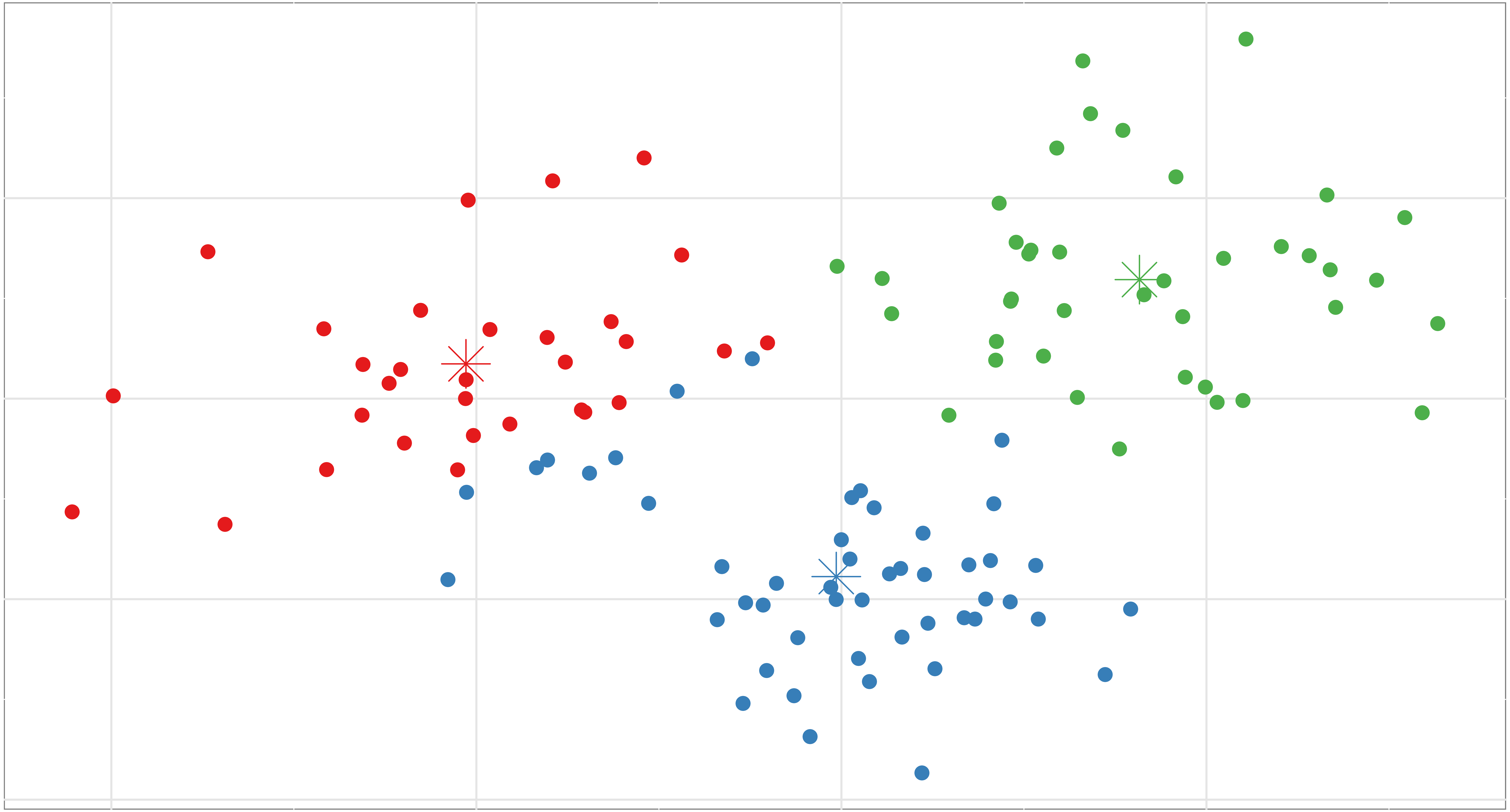
# Step 3: Move centroid to center of assigned points (1)



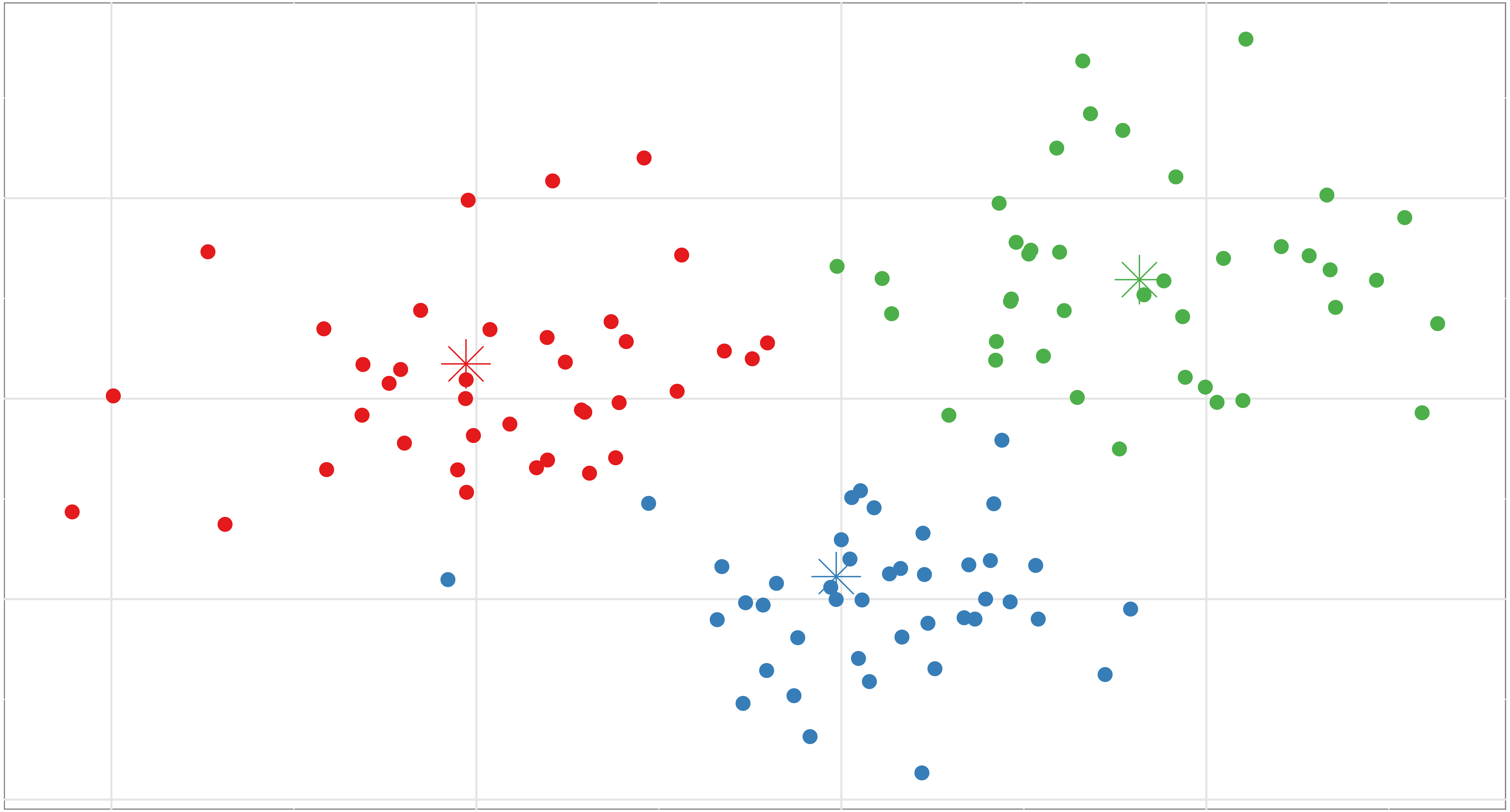
# Step 2: Assign each data point to the nearest centroid (2)



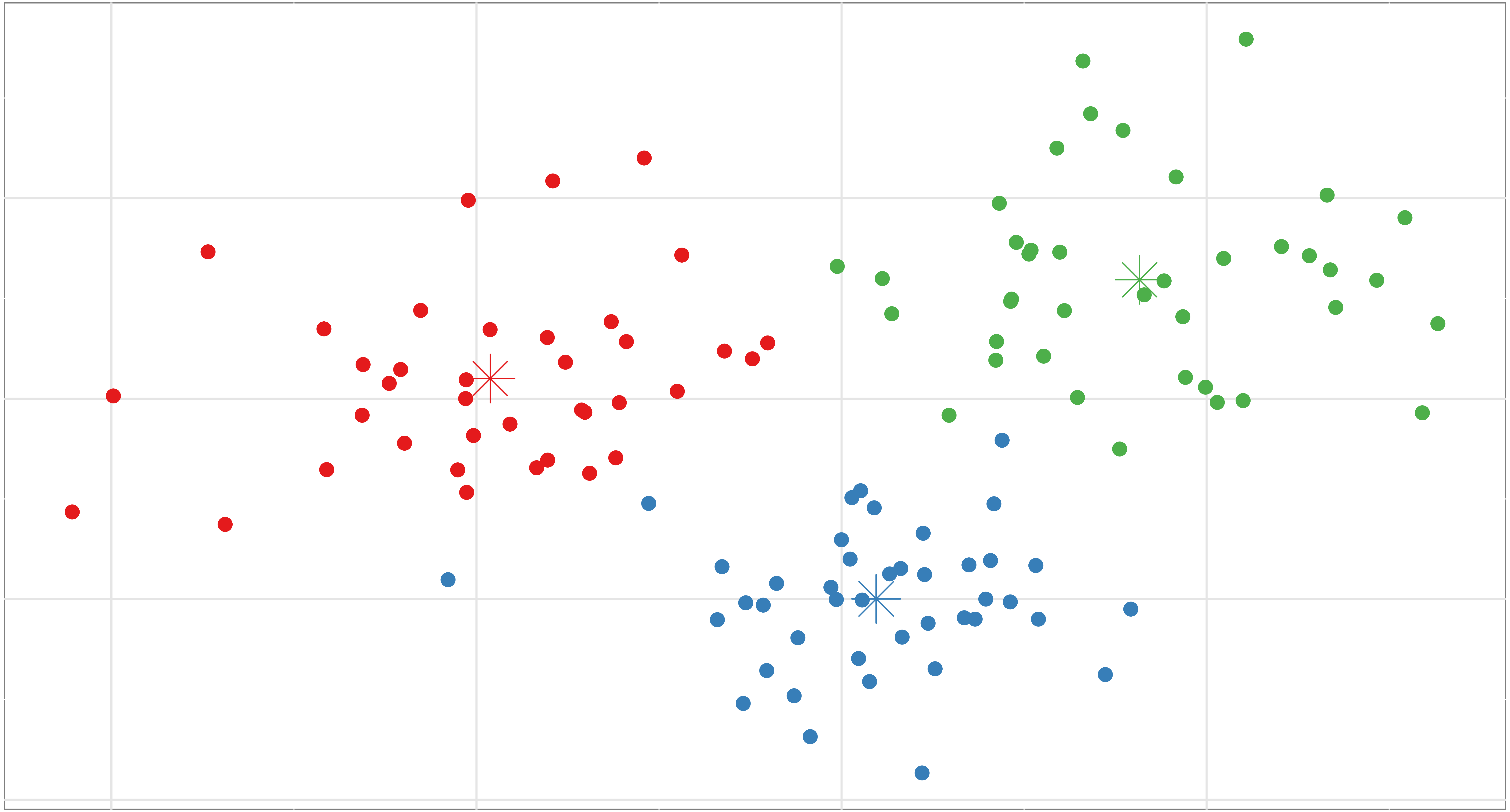
# Step 3: Move centroid to center of assigned points (2)



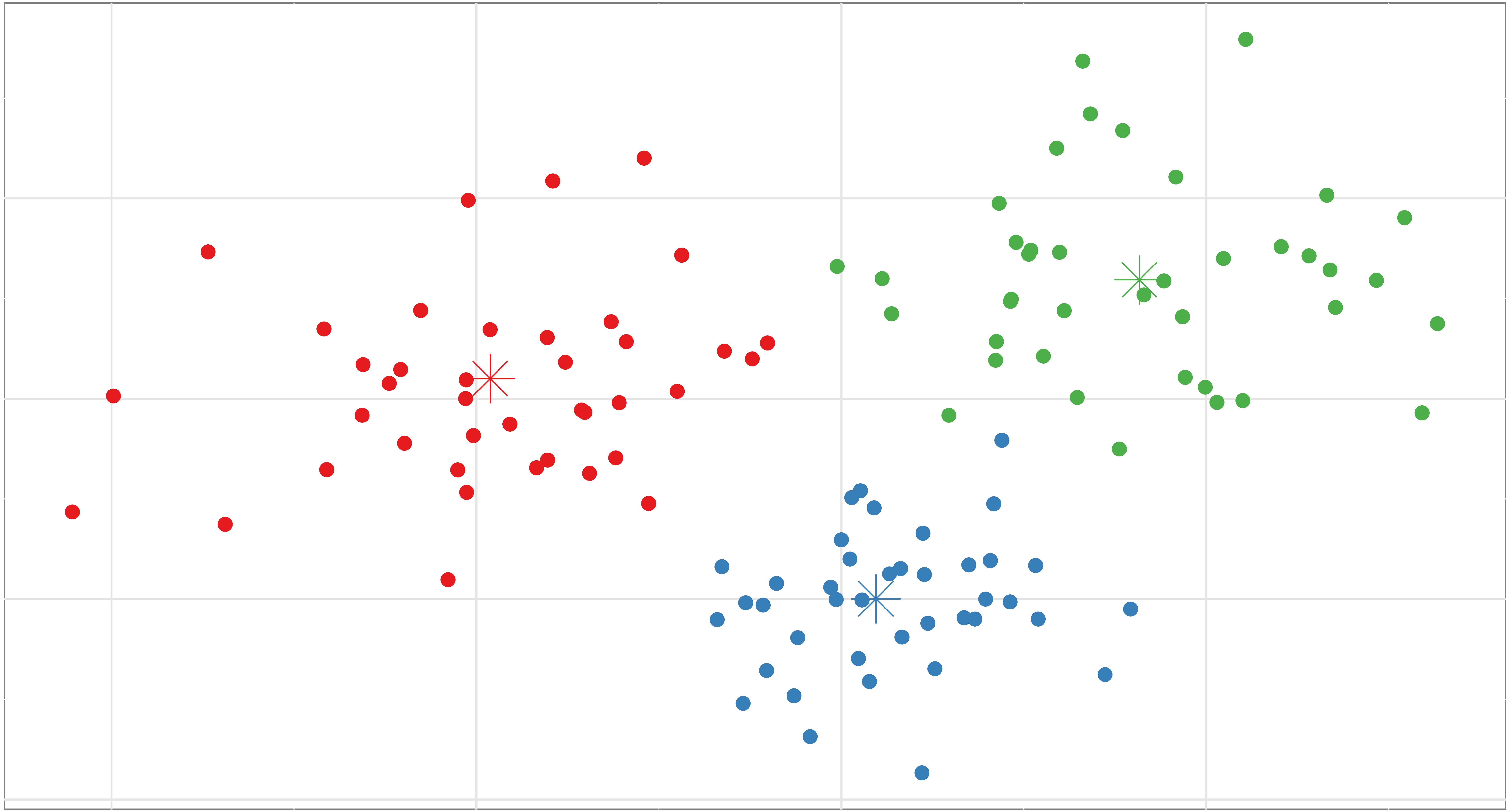
# Step 2: Assign each data point to the nearest centroid (3)



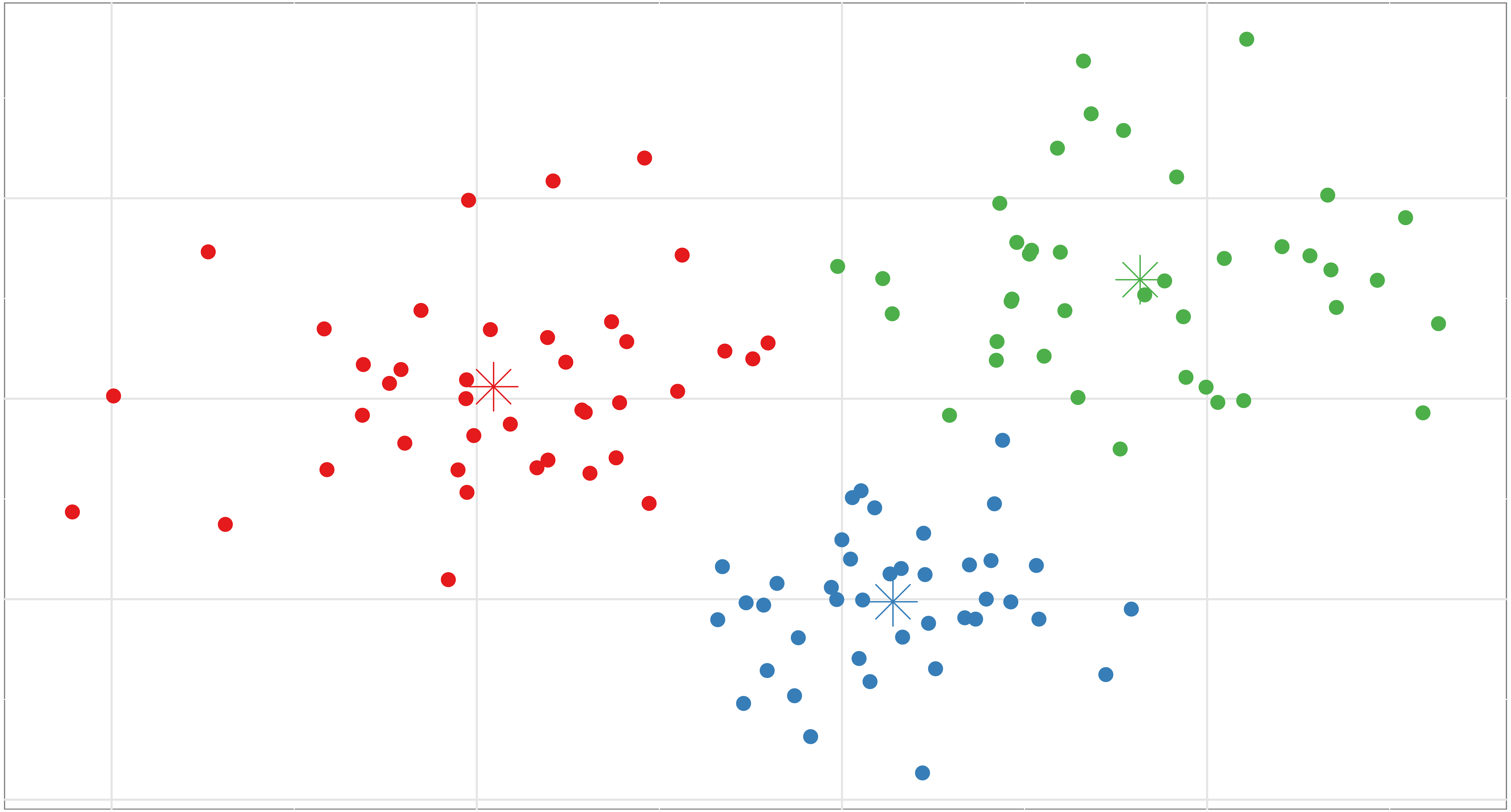
# Step 3: Move centroid to center of assigned points (3)



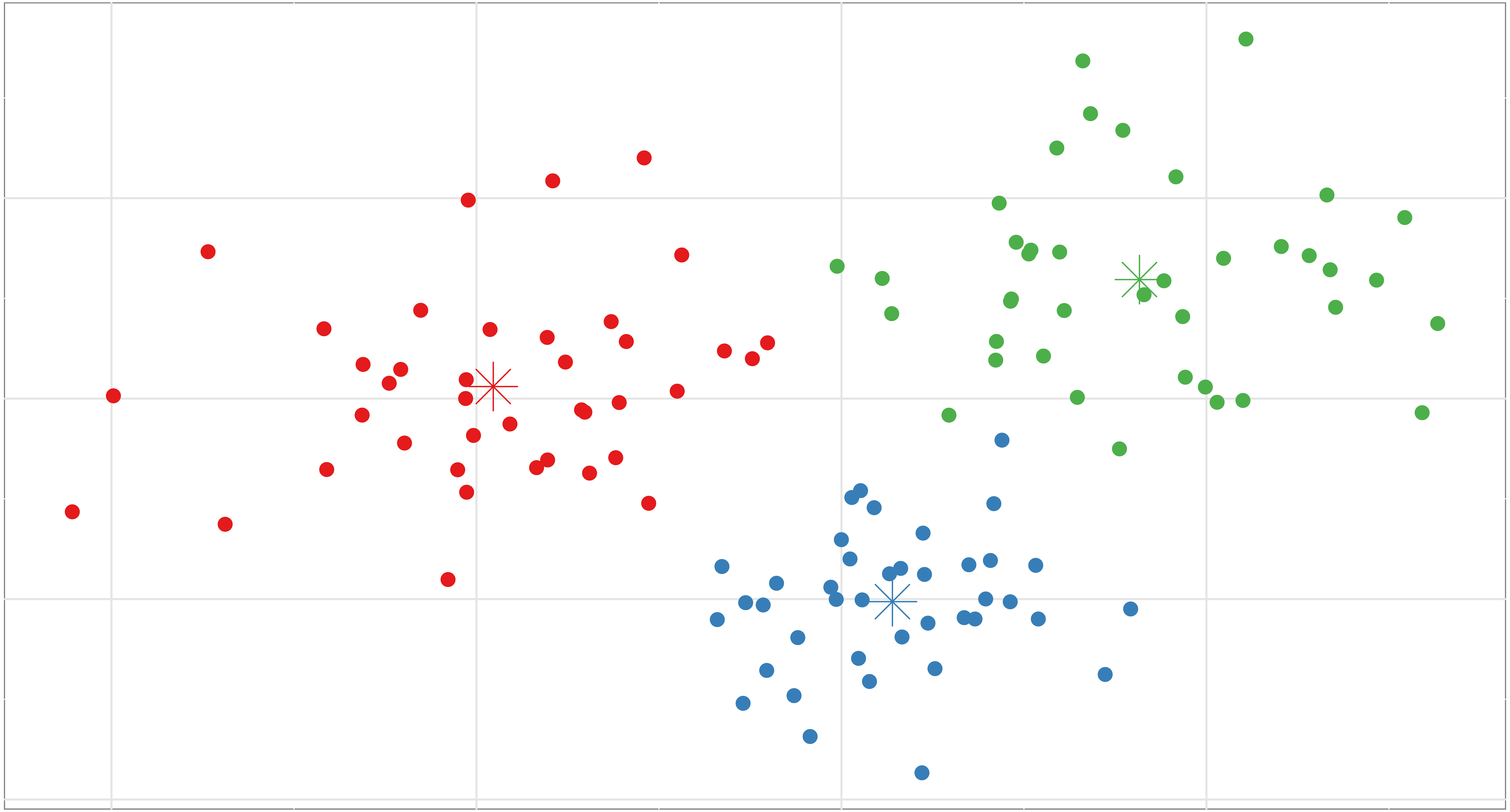
# Step 2: Assign each data point to the nearest centroid (4)



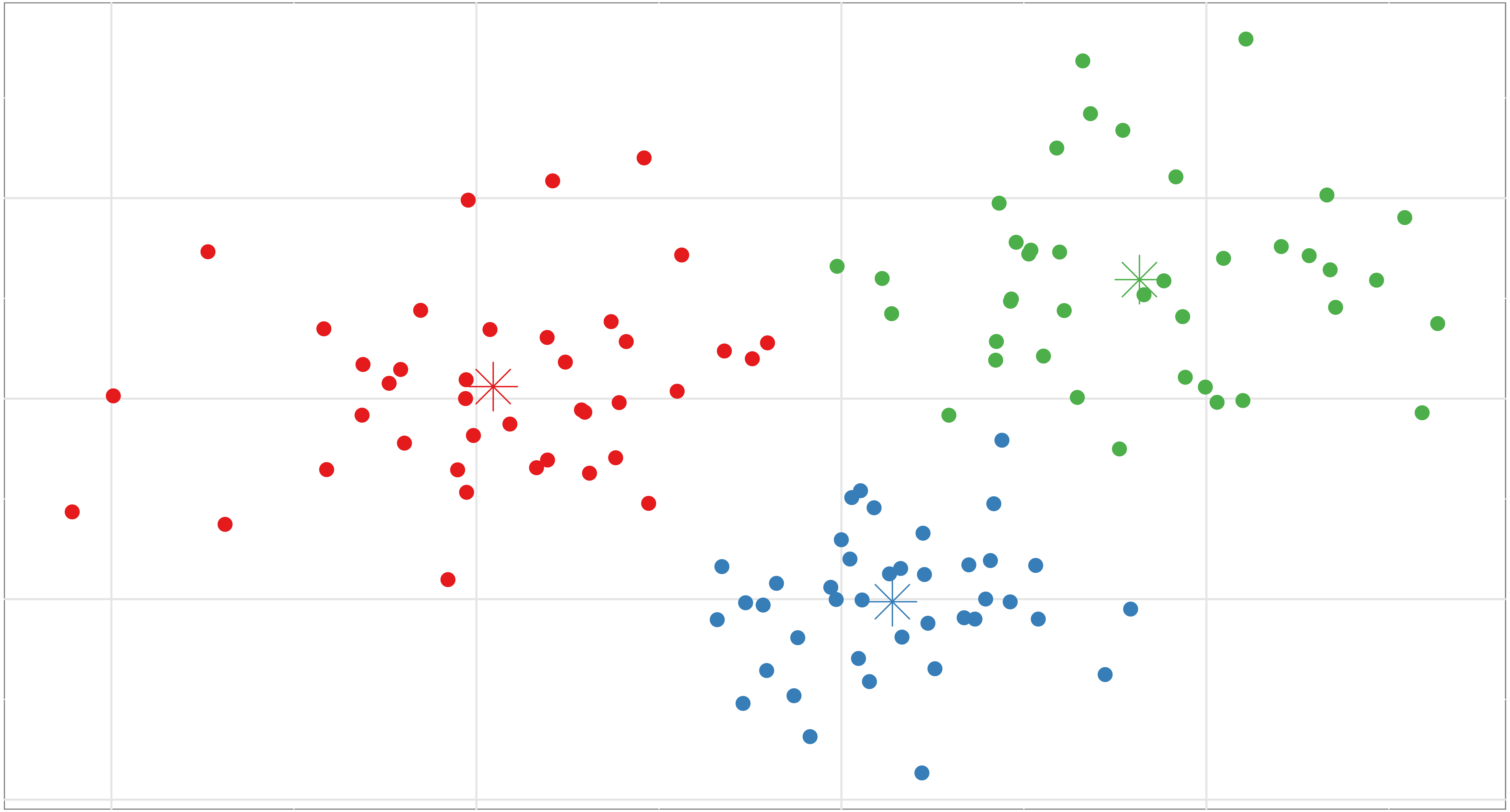
# Step 3: Move centroid to center of assigned points (4)



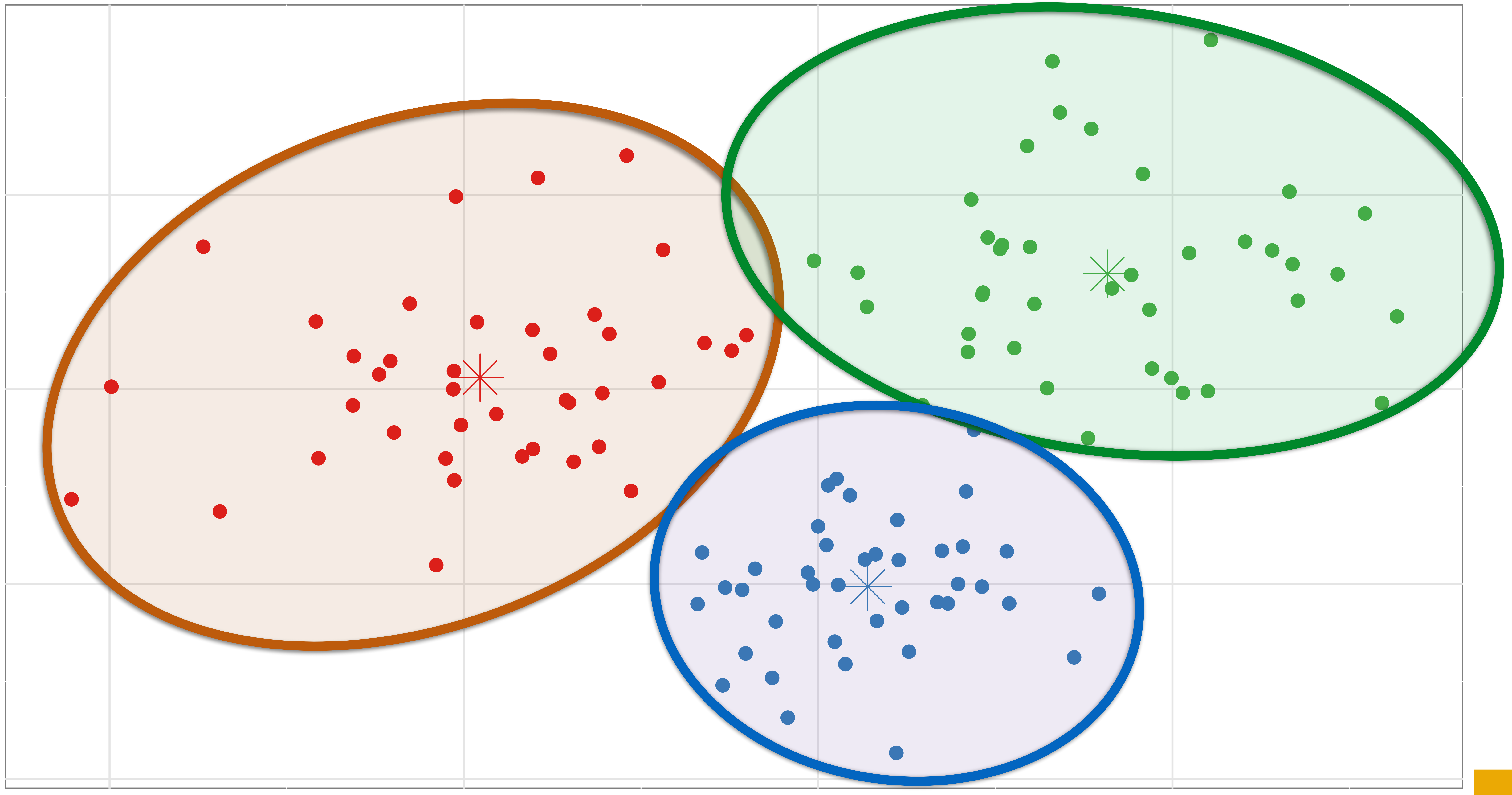
# Step 2: Assign each data point to the nearest centroid (5)



# Step 3: Move centroid to center of assigned points (5)



# Step 3: Move centroid to center of assigned points (5)



# Stop

# K-Means: Got a problem with it?

Before starting, pick the number of clusters, K

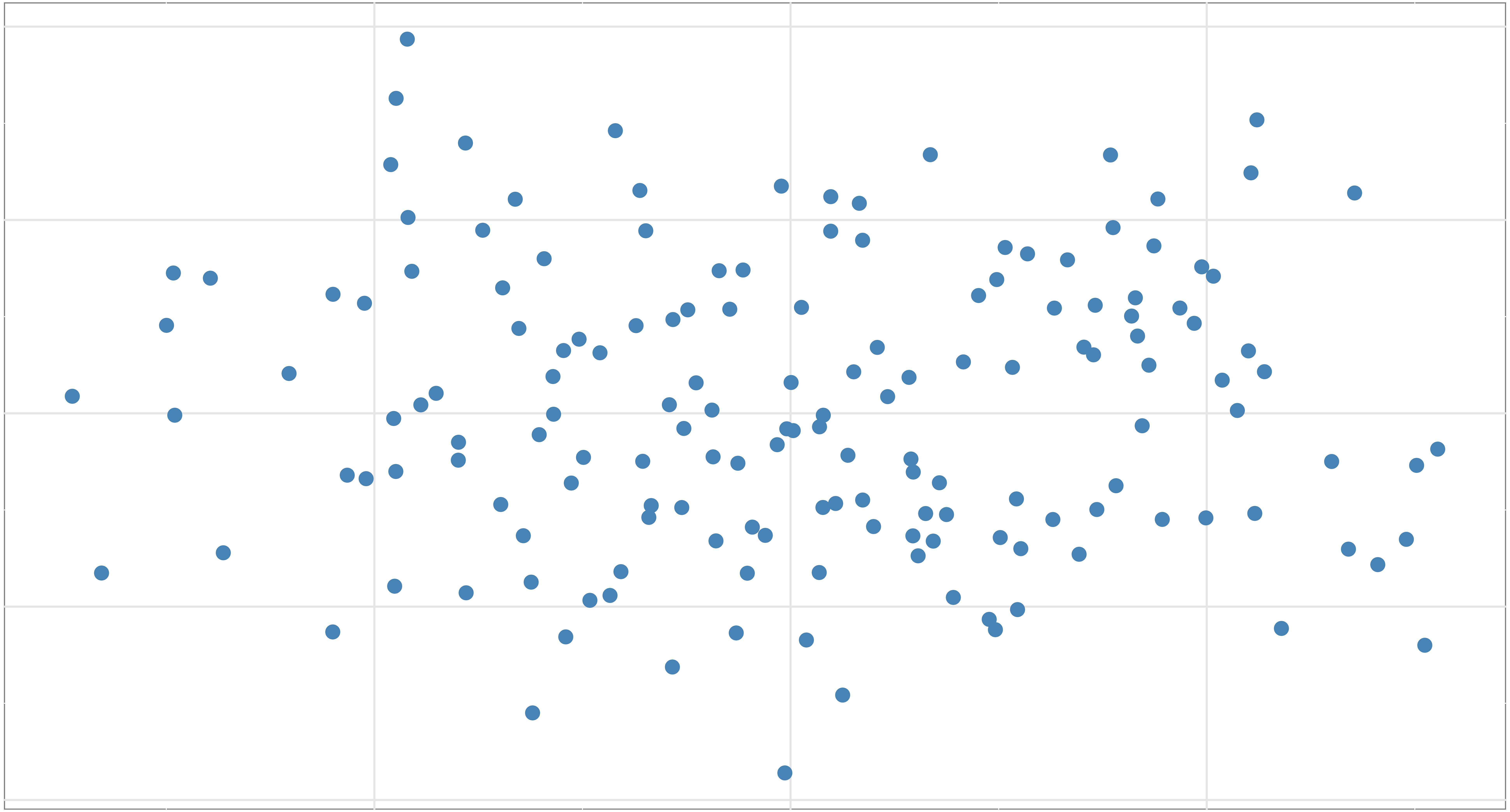
1. Pick K random centroids within data range
2. Assign each data point to the nearest centroid
3. Move centroid to center of assigned points
4. Repeat steps 2 and 3 until centroid stops shifting

# K-Means: Got a problem with it?

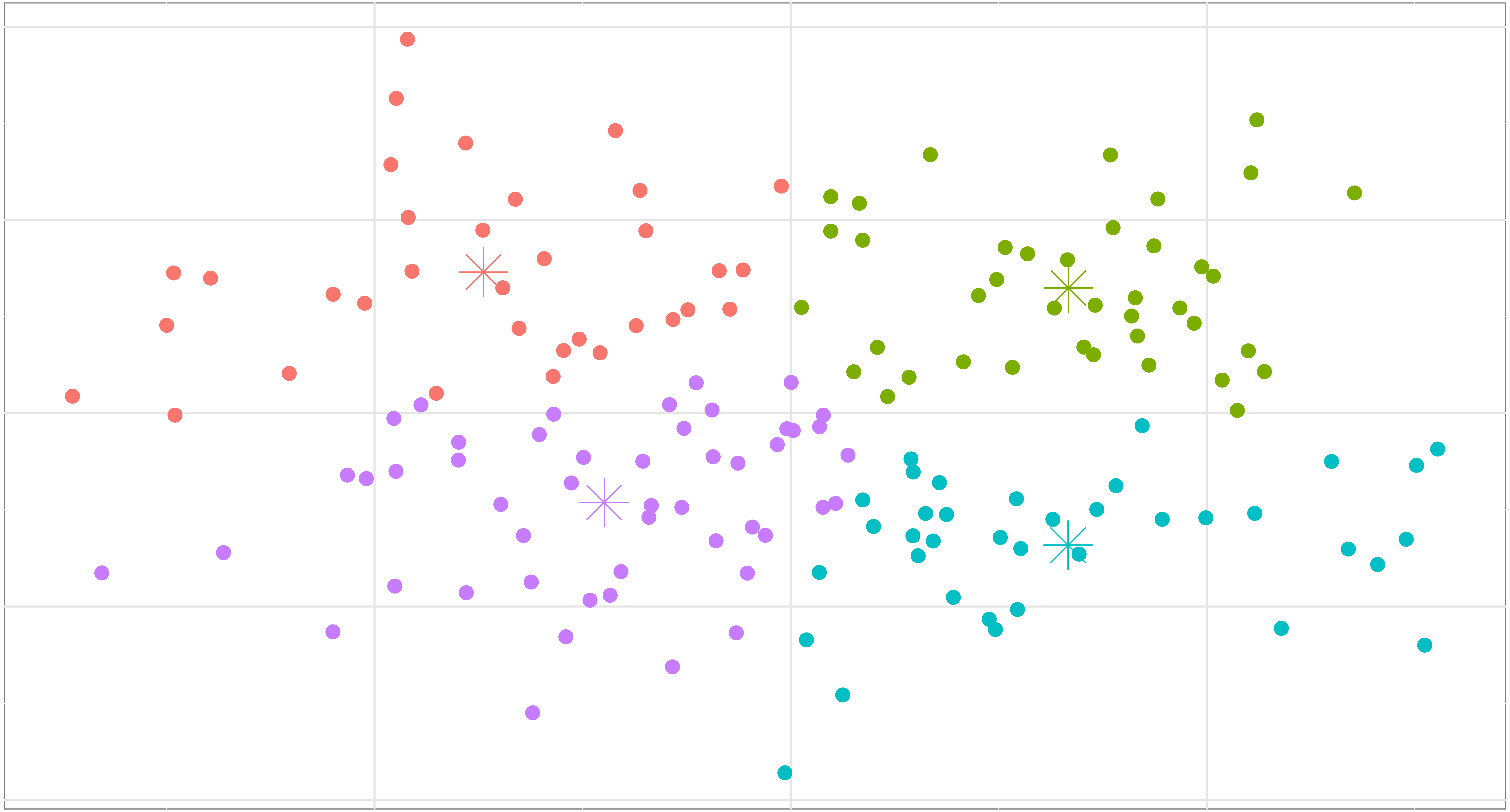
Before starting, pick the number of clusters, K Subjective

1. Pick K random centroids within data range Not Repeatable
2. Assign each data point to the nearest centroid Sensitive to Scale
3. Move centroid to center of assigned points
4. Repeat steps 2 and 3 until centroid stops shifting

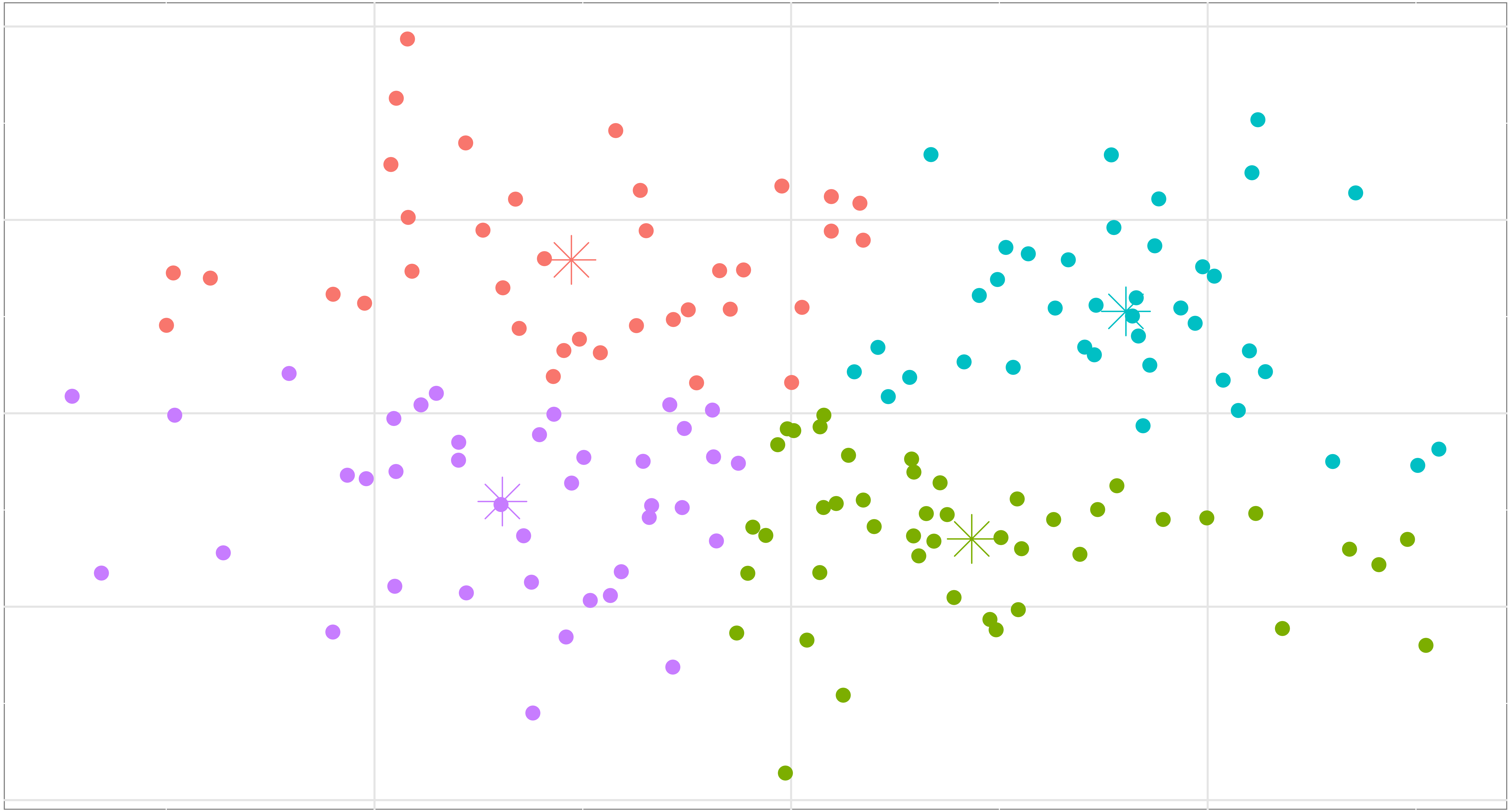
# How many clusters?



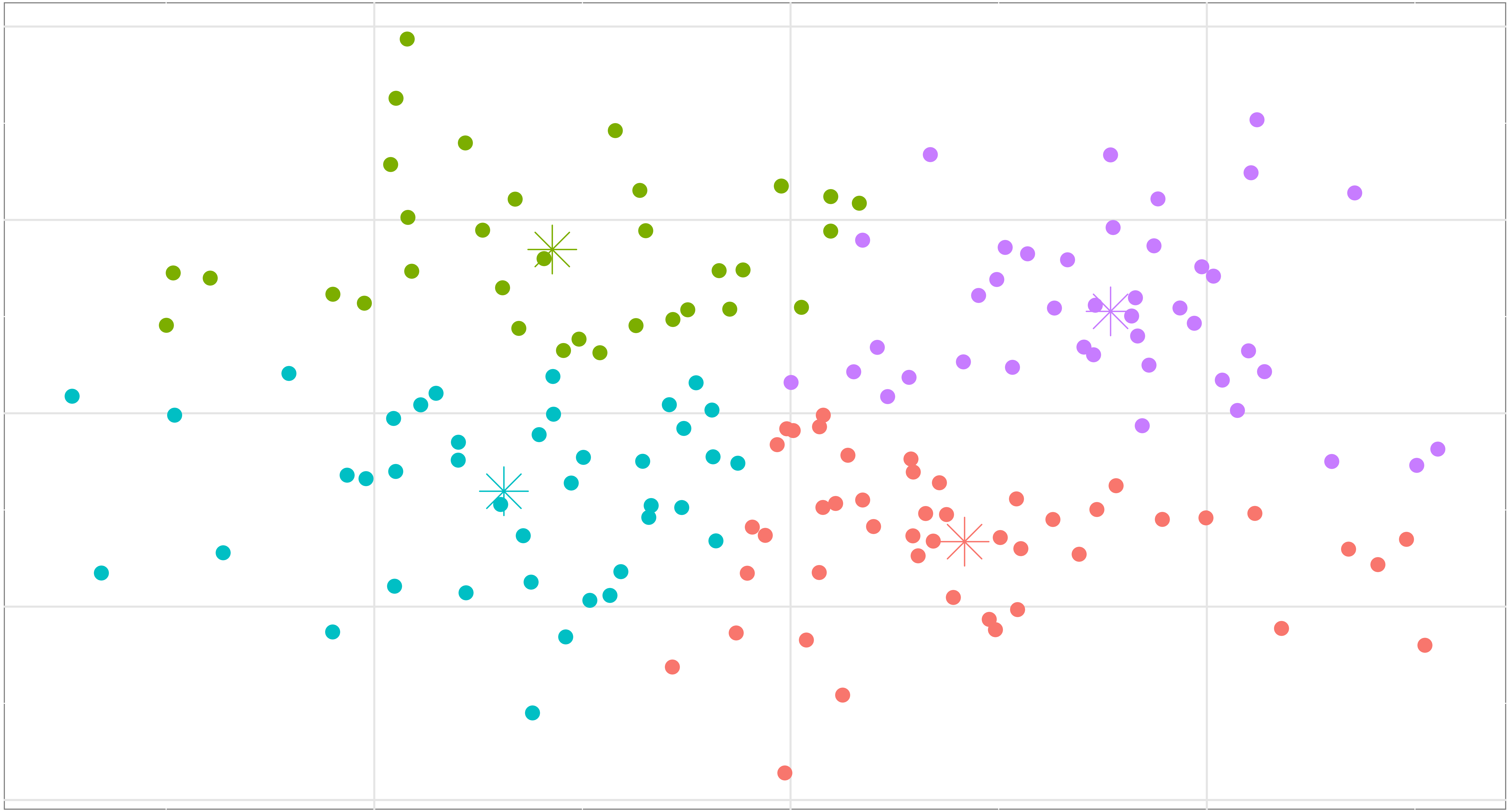
# Random Start…



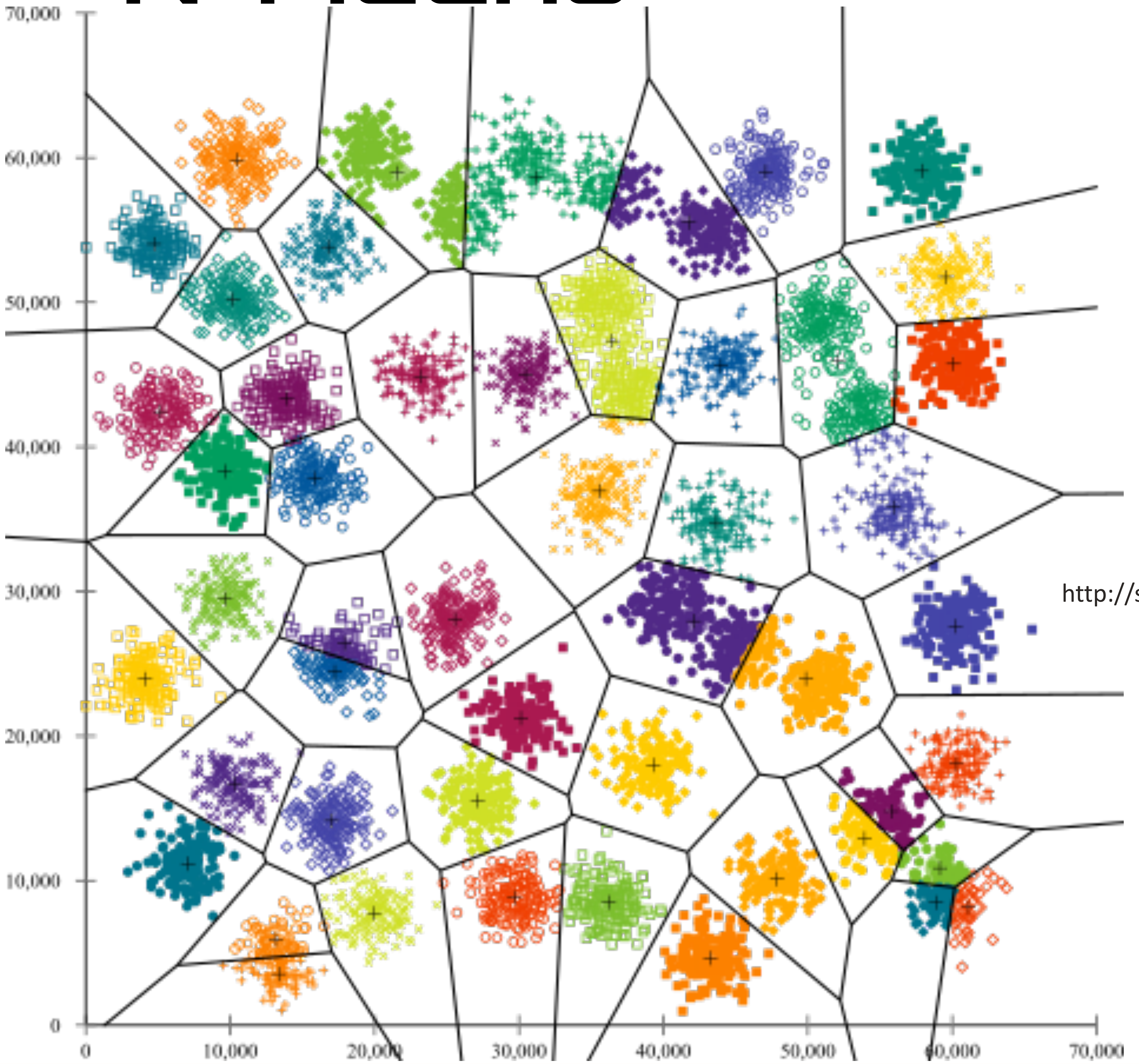
# Random Start...



# Random Start...



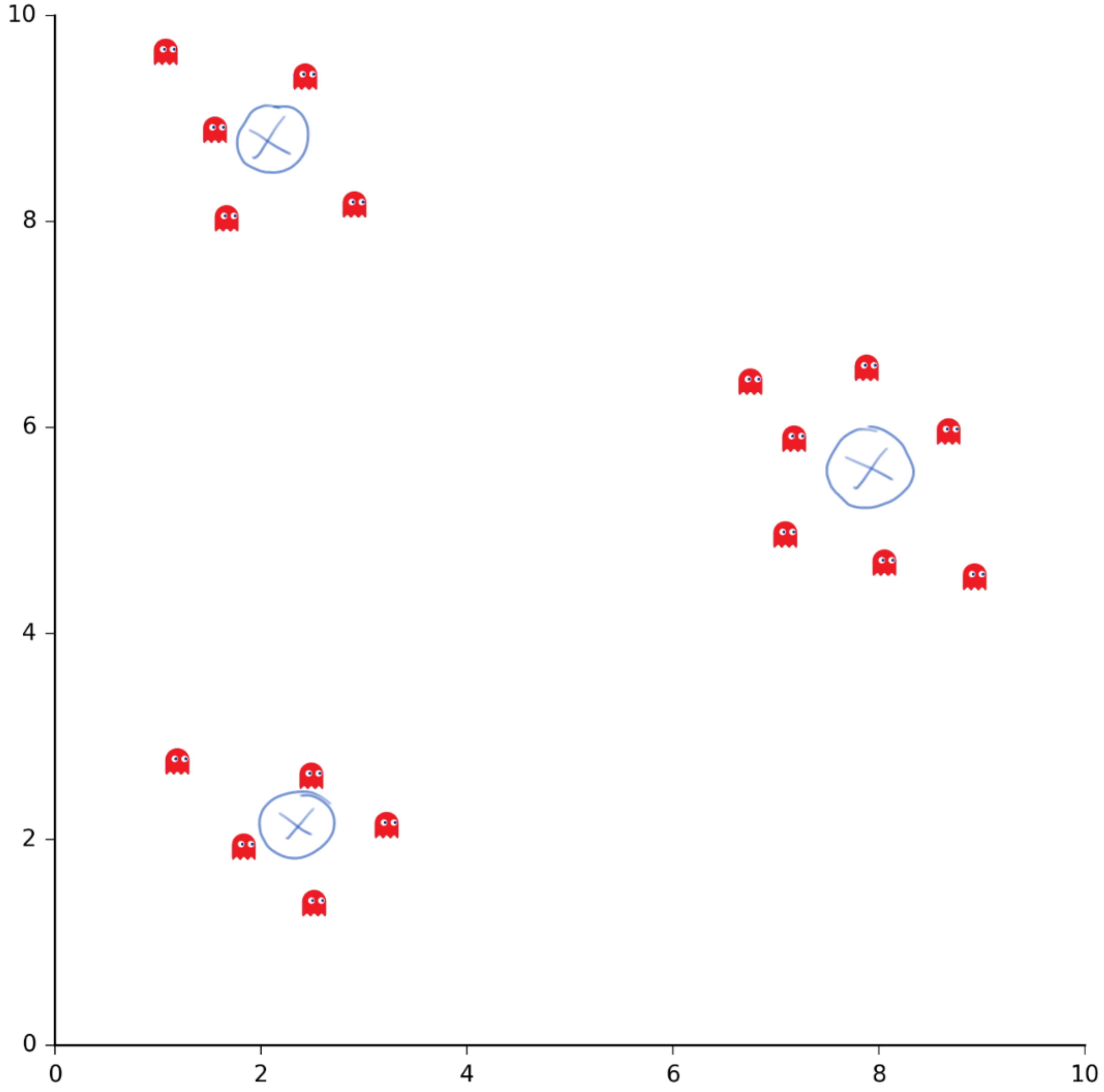
# K-Means



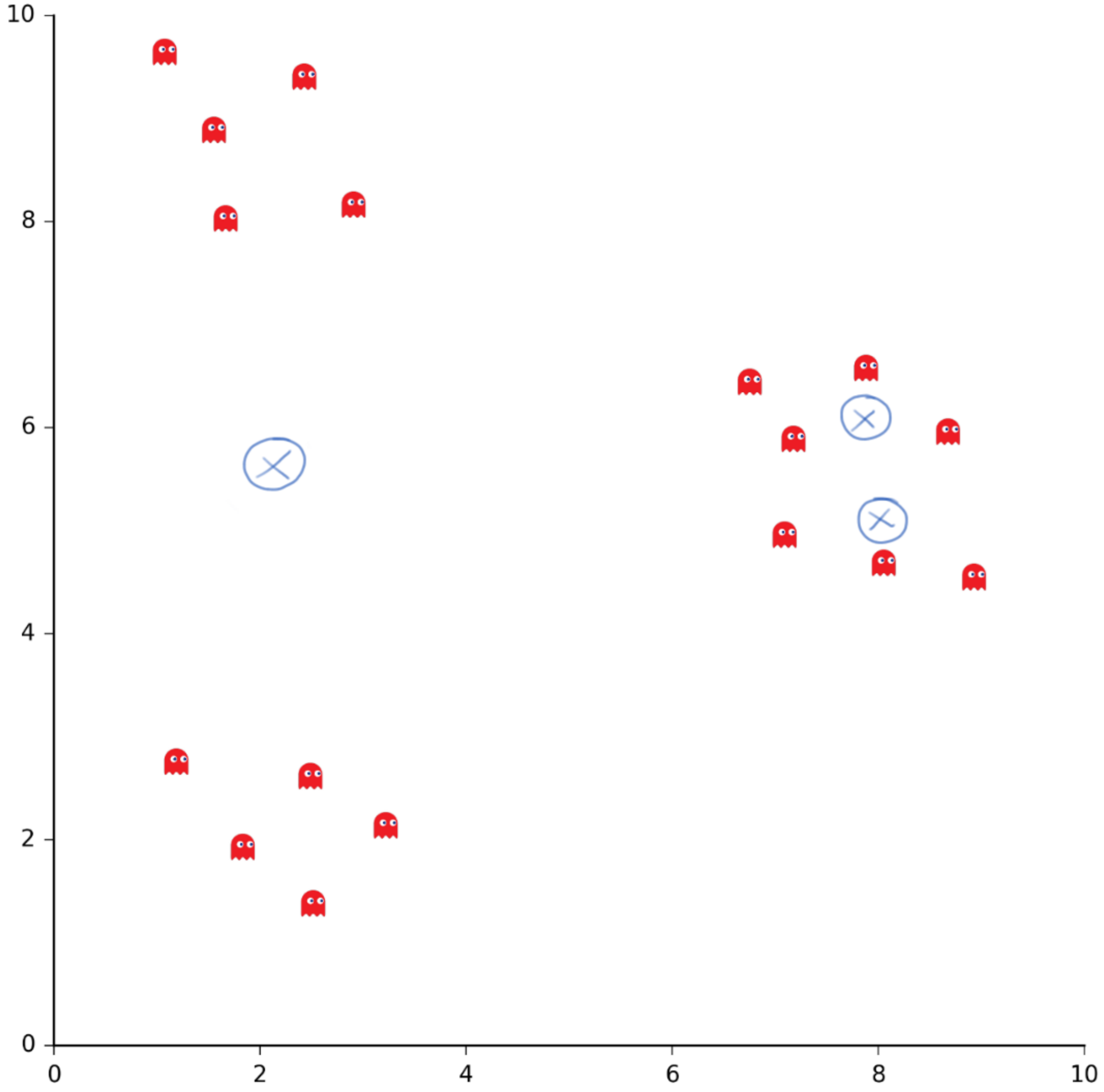

"…it's too easy to throw k-means on your data, and nevertheless get a result out (that is pretty much random, but you won't notice)."
— Anony-Mousse

http://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means

# Which one is correct?

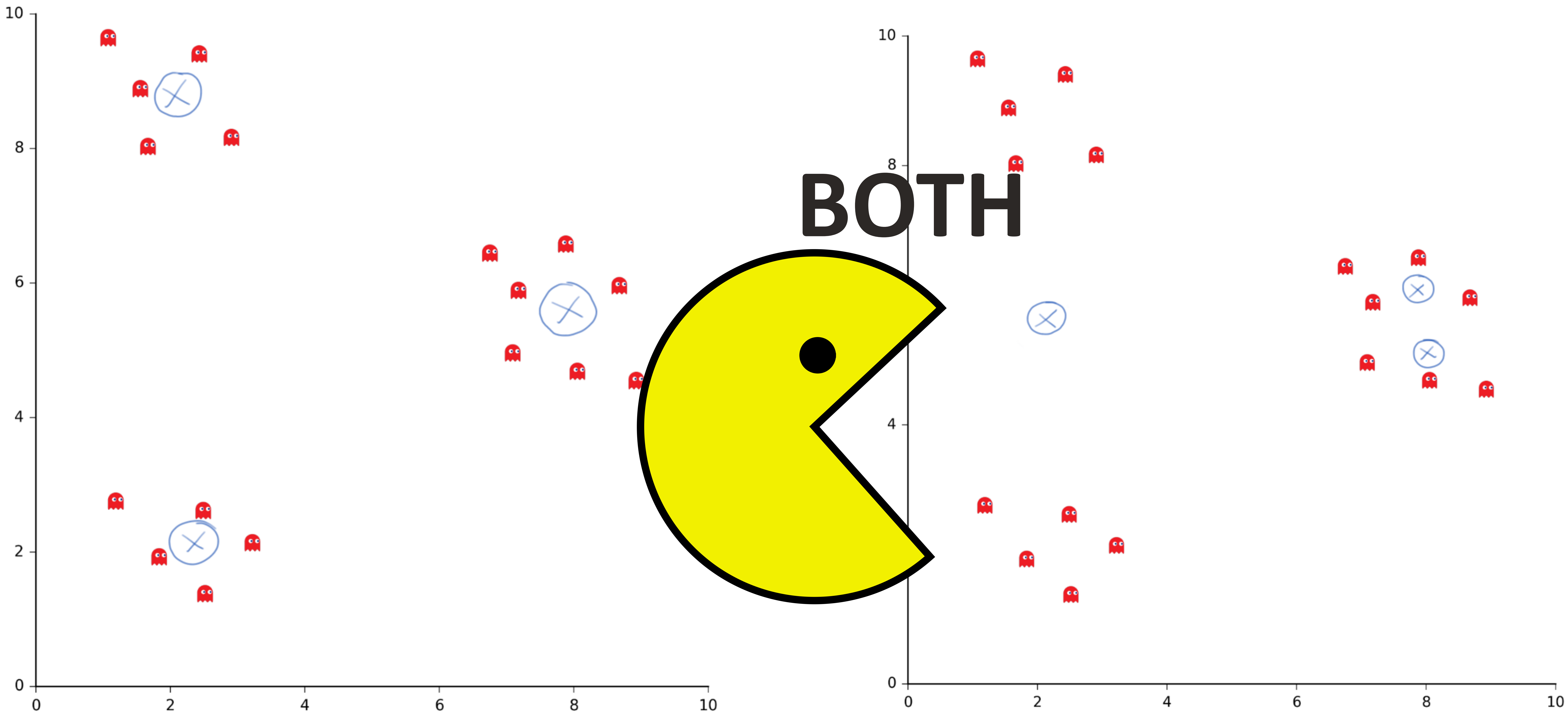



A

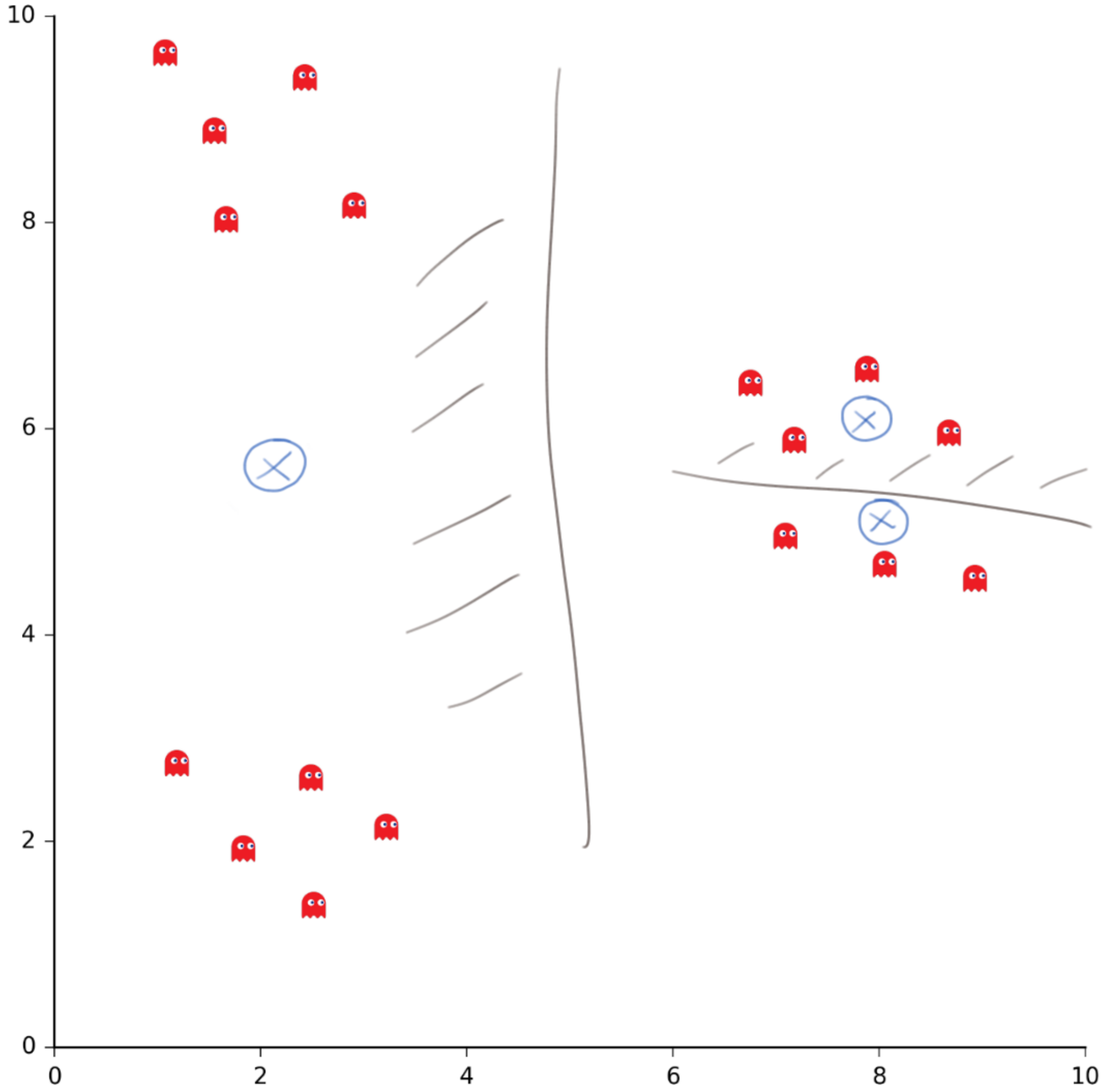



B

# Which one is correct?



# Pain of optimization…Being stuck at sub-optimal local minimum…

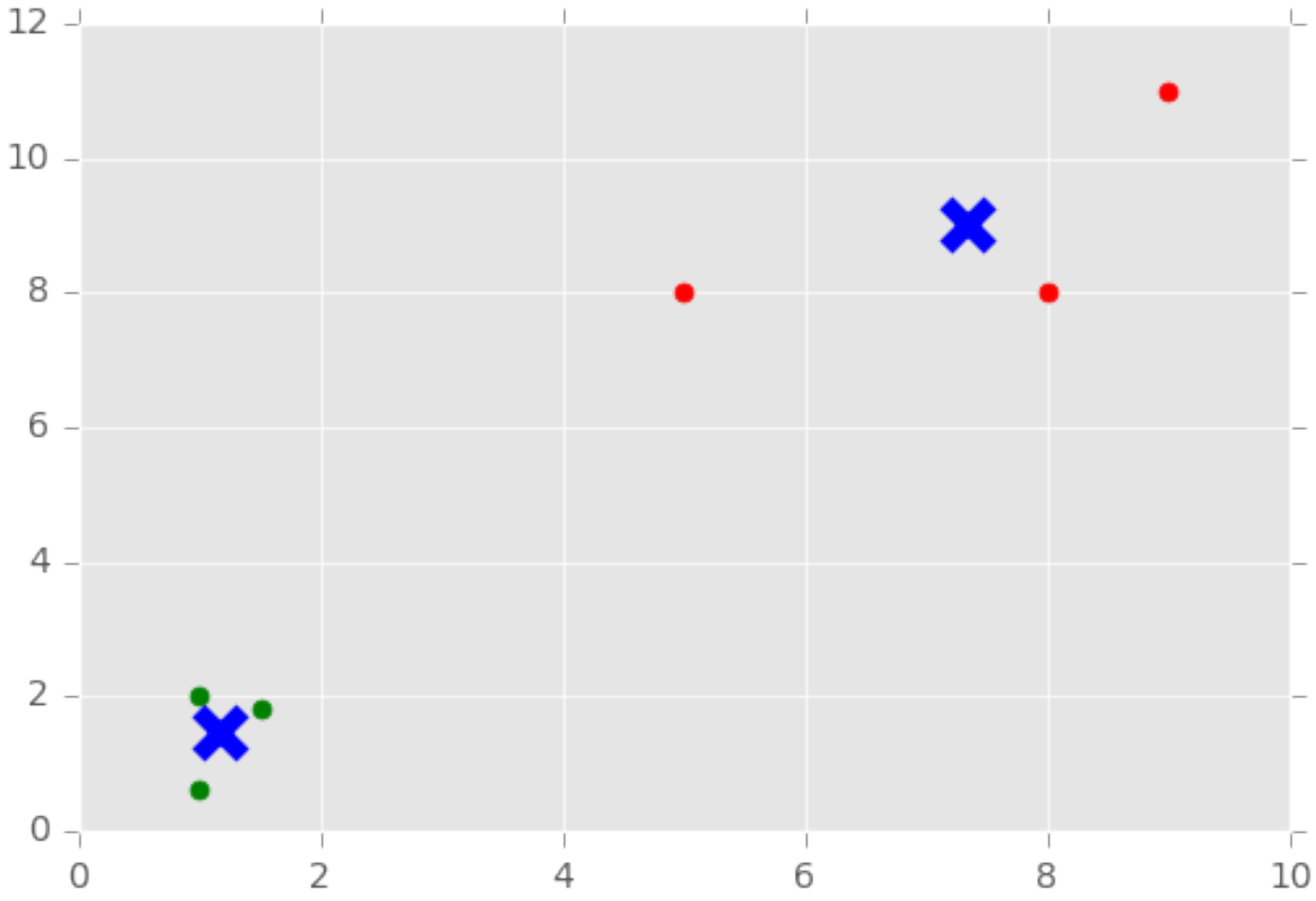



Initial guess matters!
Same outcome cannot be guaranteed

# Stop

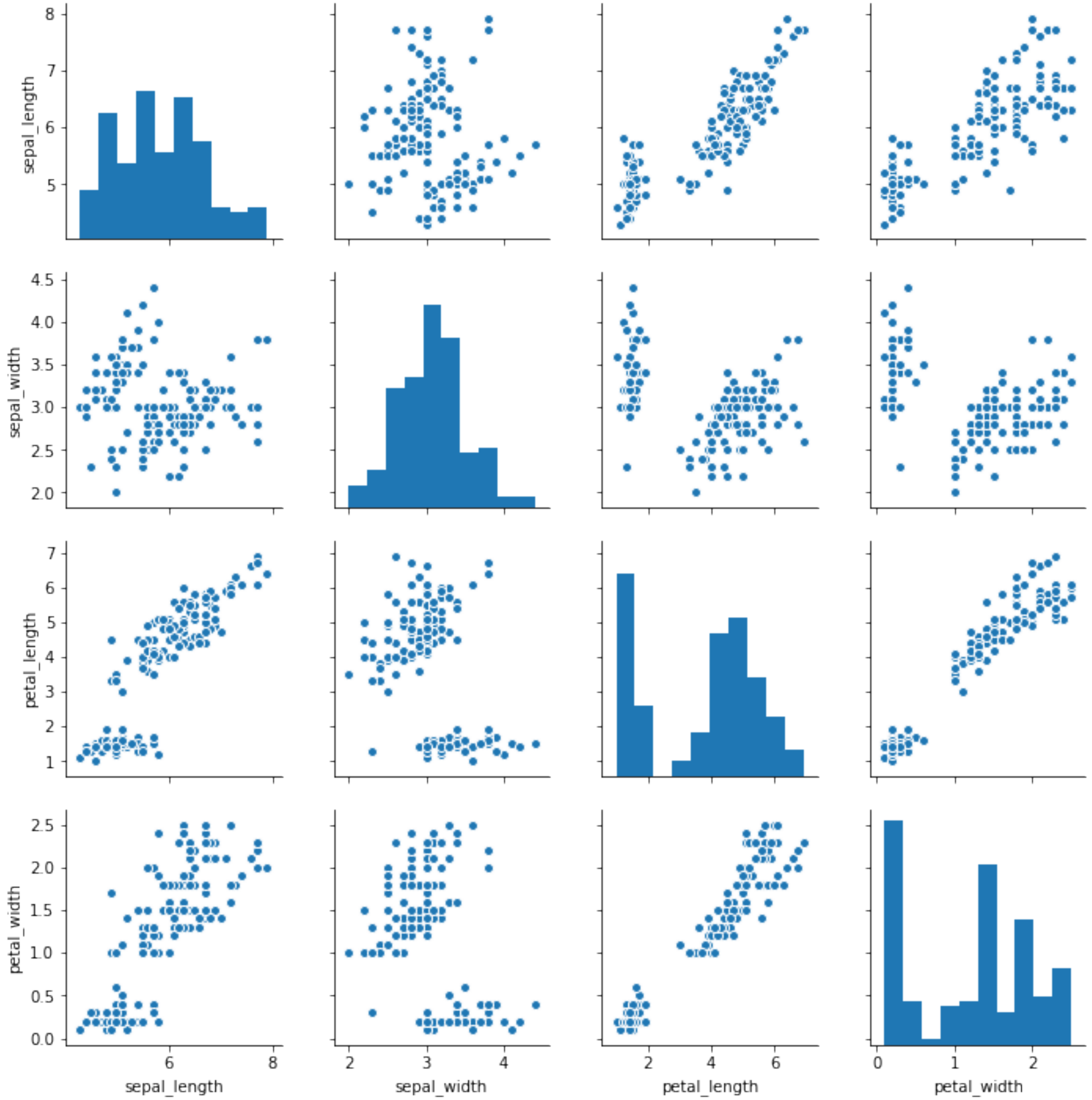
# K-Means in practice (Python version)

```
#Import from Scikit-learn
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

centroids = kmeans.cluster_centers_
labels = kmeans.labels_f
```



# The Dataset

```
sns.pairplot(<data>)
```

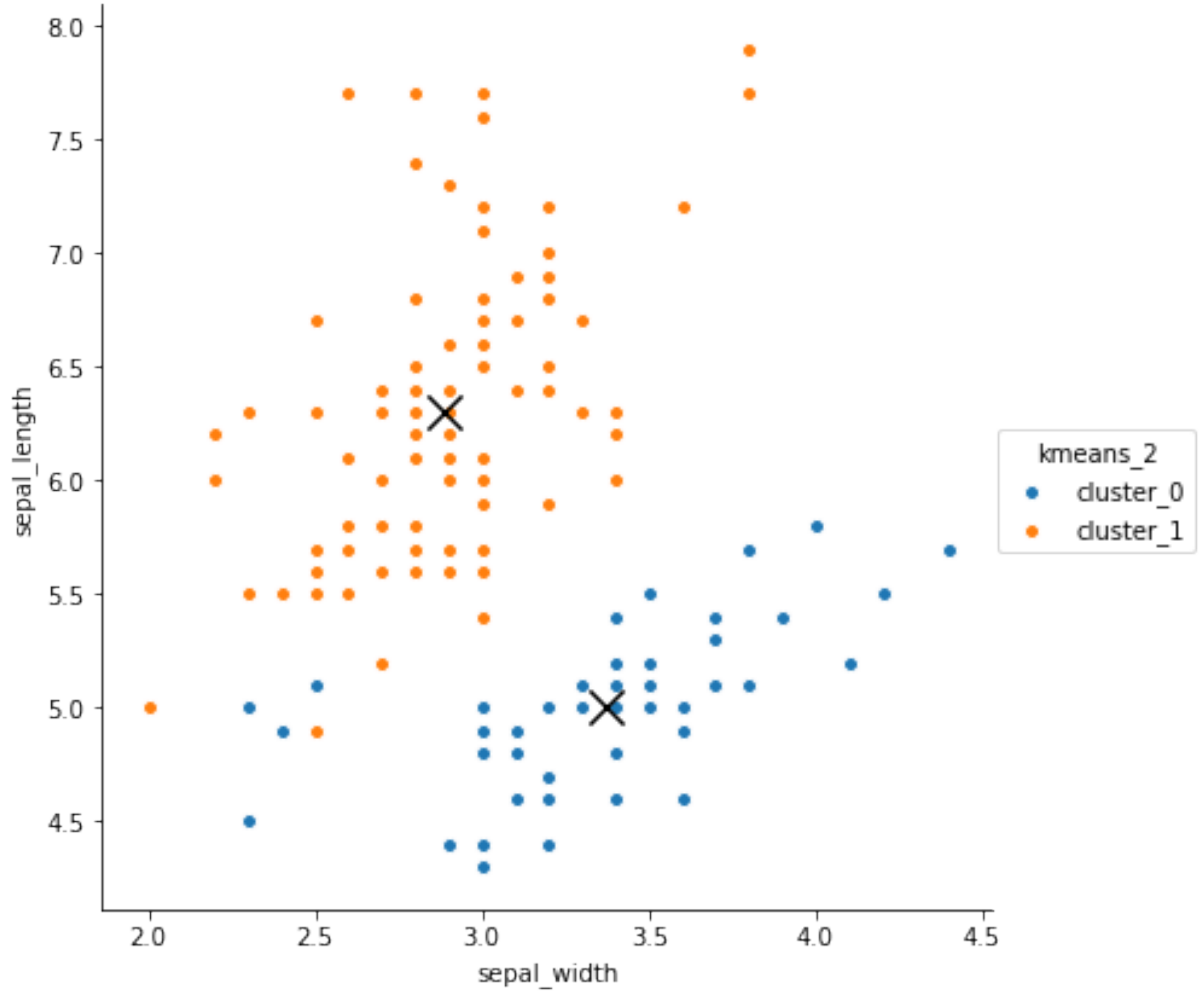


# K-Means Clustering

```
kmeans = KMeans( n_clusters=2 )
kmeans.fit( <data> )
```

# K-Means Clustering

```
sns.pairplot(<data>,x_vars="col_1",y_vars="col_2",hue="kmeans_2",size=6)
plt.scatter(<cluster_centers>,<col_2>, linewidths=3, marker='x', s=200,
c='black')
```



K-Means is affected by the scale of every feature.

# Feature Scaling

For k-means clustering, features must be scaled to the same ranges of values to contribute "equally" to the euclidean distance calculation.

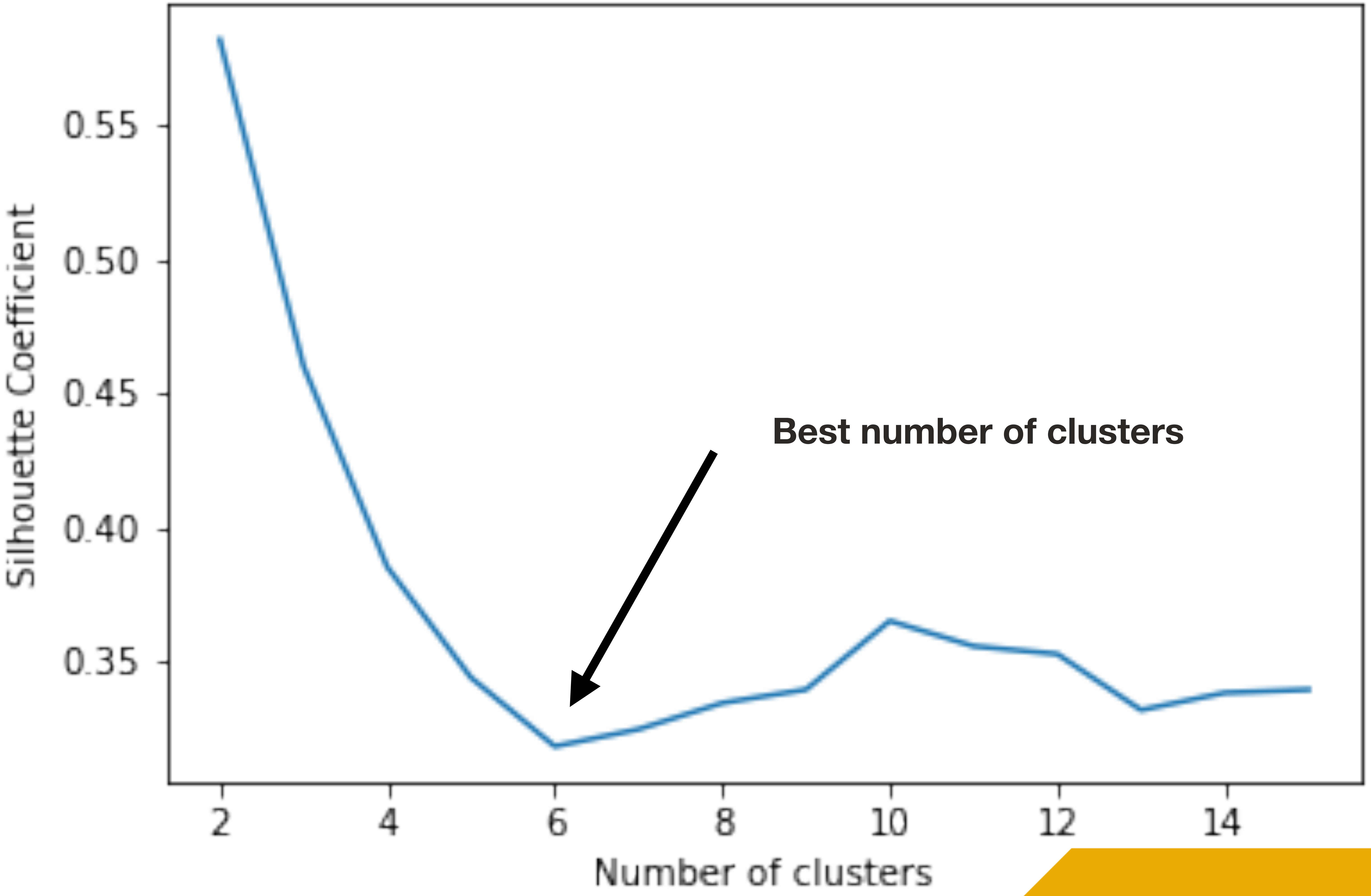
Each row is transformed per-column by:

- Subtracting from the element in each row the mean for each feature (column) and then taking this value and
- Dividing by that feature's (column's) standard deviation.

# Evaluating your Model

```
WCSS=[]
for i in range(1,20):
  kmeans=KMeans(n_clusters=I, init='k-means++')

kmeans.fit(final_data[scaled_feature_columns].sample(50000))
  WCSS.append(kmeans.inertia_)
```

# Evaluating your Model



# Evaluating your Model

```
from yellowbrick.cluster import SilhouetteVisualizer
model = MiniBatchKMeans(6)
visualizer = SilhouetteVisualizer(model)

visualizer.fit(X)
visualizer.poof()
```