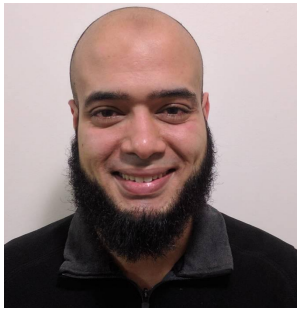
Machine Learning

K-Means

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

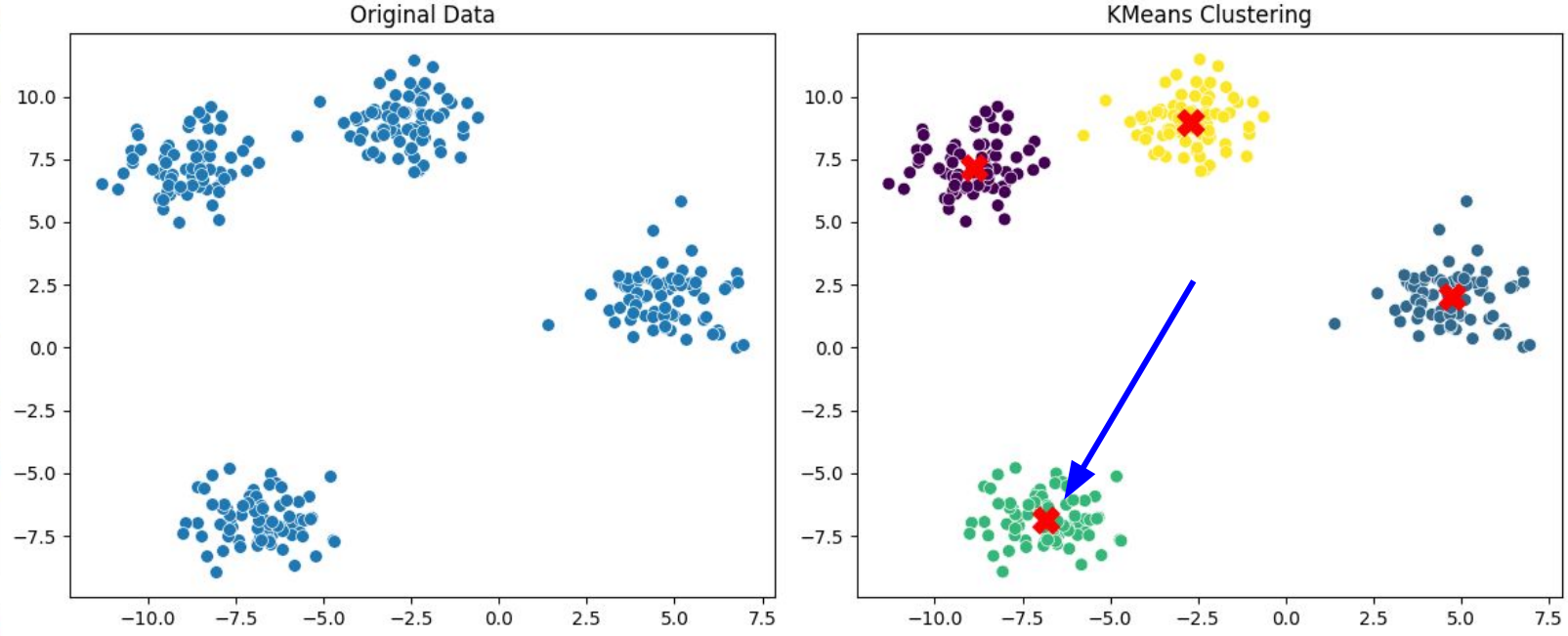


Clustering

- **Clustering** is a method of **unsupervised** learning in which data points are grouped together based on their similarities.
- The goal is to partition a dataset into **groups** (clusters) where items within the same group are more **similar** to each other **than** to those in other groups.

Clustering

- Below data coming from 4 groups (no annotation)
- K-Means algorithm will identify the groups and provide a **center** for each



Clustering Types

- Centroid-based
  - **K-Means**: simple algorithm and most widely used (always try first)
  - **K-Medoids**: variant of K-means
- Hierarchical Clustering: Groups data over **multiple levels** of clusters
  - Agglomerative (bottom-up) and Divisive (top-down)
  - Think data has levels (e.g. car vs honda - tree (body) vs branch (part)
- Density-based Clustering
  - Find Dense regions of data points separated by regions of lower point density
  - DBSCAN, OPTICS, HDBSCAN
- Graph-based Clustering
- Gaussian Mixture Models (GMM)
  - Uses expectation-maximization to fit multiple Gaussian distributions to the data

K-Means

- Given K (number of the target clusters), partition a dataset into K groups where the data points in each cluster are closer to each other than they are to points in other clusters
  - A point is assigned only to one group (cluster)
- Each cluster can be represented by a centroid: the average of all the cluster’s points

Steps of the K-Means Algorithm

- **Initialization**: Randomly select K data points from the dataset as the initial centroids
- **Assignment**: Assign each data point to the nearest centroid.
  - This forms K clusters
- **Update**: Calculate the mean of all the points for each cluster and set the centroid to this mean
- **Repeat**: Continue repeating the process until the centroids no longer change significantly

Data Structure

- There are different ways to implement K-Means
- You need first to decide what to store
  - Point perspective: for each point state its cluster: 1 to 1
  - **Cluster perspective**: for each cluster state its points: 1:M
    - More efficient

```
class KMeansCustom:
    def __init__(self, k=3, max_iter=100):
        self.k = k
        self.max_iter = max_iter
        # dict of indices [0-K-1]: for each cluster idx: list of its points
        self.clusters = {}
        # List of indices [0-K-1]: for each cluster idx: D-dimensional vector for centroid
        self.centroids = []
```

K-means: Utils

- First function just help us find initial centroid
  - We pick random points
- Second one compute the distance between point and cluster centroid

```
def initialize_centroids(self, data):
    # select k random different centroids: assumes k <= # number of examples
    random_indices = np.random.choice(data.shape[0], self.k, replace=False)
    self.centroids = data[random_indices]

def _dist(self, point, centroid):
    #return np.linalg.norm(point - centroid)     # has sqrt
    return np.sum((point - centroid)**2)        # we don't need actual distance
```

K-Means: Assign and Update

- The heart of the algorithm is to
  - For each point find the nearest cluster.
    - This way, we identify for each cluster its points
  - Once we have the points per cluster, just average them

```
def assign_clusters(self, data):
    self.clusters = {i: [] for i in range(self.k)}
    for point in data:
        distances = [self._dist(point, centroid) for centroid in self.centroids]
        cluster_idx = np.argmin(distances)
        self.clusters[cluster_idx].append(point)    # add cluster points

def update_centroids(self):
    for cluster_idx, cluster_points in self.clusters.items():
        # Average points of each cluster. axis=0 ==> vertically
        self.centroids[cluster_idx] = np.mean(cluster_points, axis=0)
```

K-Means: Fitting

- To fit, we iterate I times
  - Then we simply keep assigning and updating the centroids
  - In the middle, we may notice that the centroids don’t change any more
    - Then we can stop for such convergence

```
def fit(self, data):
    self.initialize_centroids(data)

    for _ in range(self.max_iter):
        self.assign_clusters(data)
        prev_centroids = np.copy(self.centroids)
        self.update_centroids()

        # Check for convergence
        if np.allclose(self.centroids, prev_centroids, rtol=1e-4):
            break
```

K-Means: Predicting

- Given a new point, we can simply find the nearest centroid for it
- This represents its cluster
- Total time complexity: O(I × N× K× D)
  - I iterations / N points / K Clusters / each cluster is D dimensions
  - Verify!

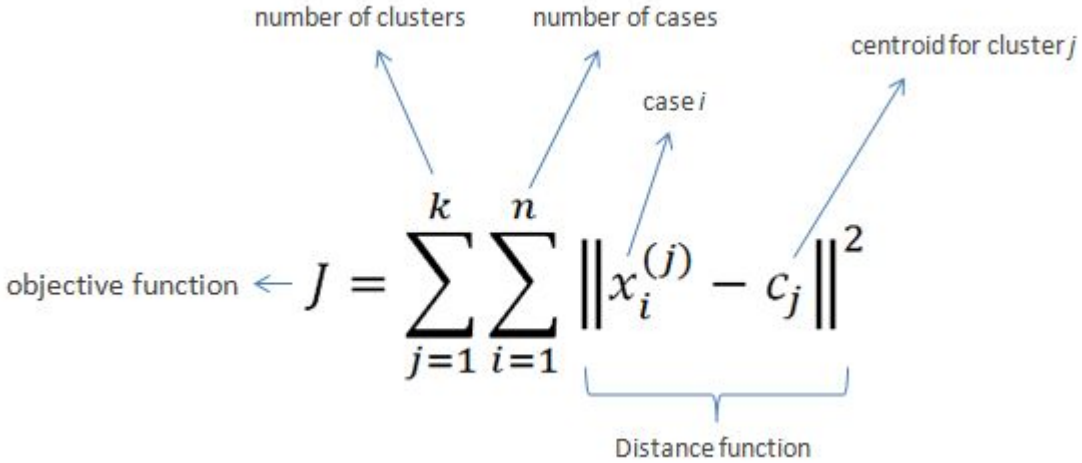
```
def predict(self, data):
    predictions = []
    for point in data:
        distances = [self._dist(point, centroid) for centroid in self.centroids]
        cluster_idx = np.argmin(distances)
        predictions.append(cluster_idx)
    return predictions
```

Keep in Mind

- You have to provide K
  - Some algorithms don’t need that
  - We may find ways to estimate an initial number
- Initialization Sensitivity
  - The algorithm can converge to a **local optimum** depending on the initialization of the centroids
  - Solution: Multiple runs with different initializations and select the minimal error
    - Define as the distance between the cluster and its points
- Others
  - You better discard outlier points
  - You better scale the data first
  - You can use other distance metrics if make more sense

Why Local Minima?

- The primary objective of the K-Means algorithm is to minimize the **within-cluster sum of squares** (WCSS), which represents the sum of squared distances between data points and their corresponding centroid
- This objective function is **non-convex** with respect to the cluster assignments and the centroids



Img src

Interview Tip

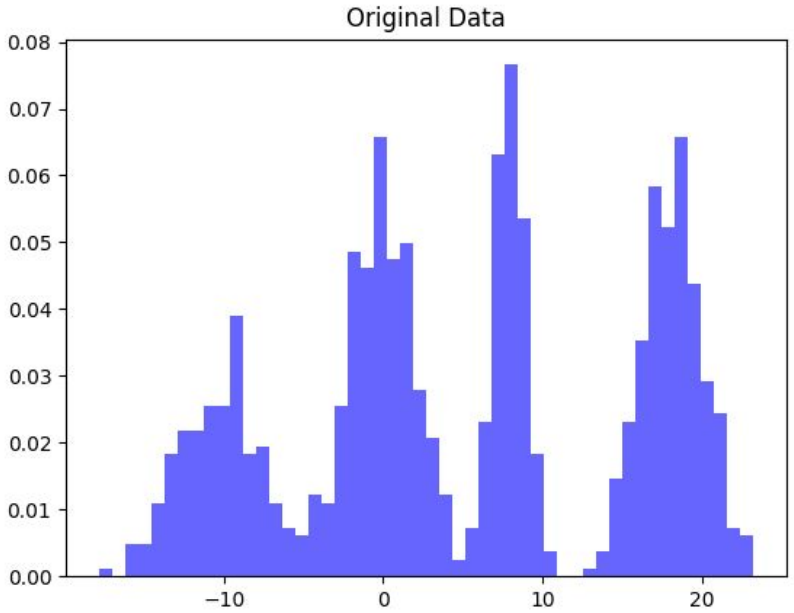
- Asking to code K-means live is a common coding interview question
- Make sure to practice it!

Gaussian Mixture Model (GMM)

- GMM is a probabilistic model for **soft** clustering
  - K-means is hard clustering: a point belongs to one specific cluster
  - Soft clustering: We get a probability of a point to belong to a cluster
- GMM assume the data is **generated** from a mixture of several Gaussian distributions
  - GMM finds a mixture of Gaussian distributions that represents this data
- It involves 2 steps similar to soft K-means steps
  - **Expectation** Step (similar to assign cluster) and **Maximization** Step (similar to update)
- Cons: computationally more expensive than K-means
- Extra usage: **Density Estimation** - GMM can also be used for estimating the density function for a dataset (generative model)

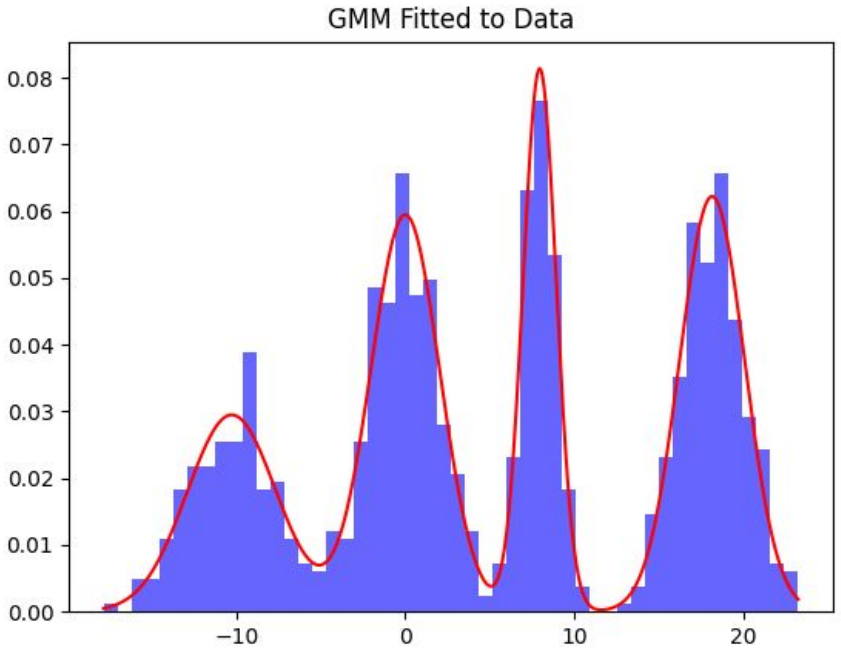
GMM

- Assume we want to represents this data as multiple gaussians
- How **many gaussian** distributions are there?



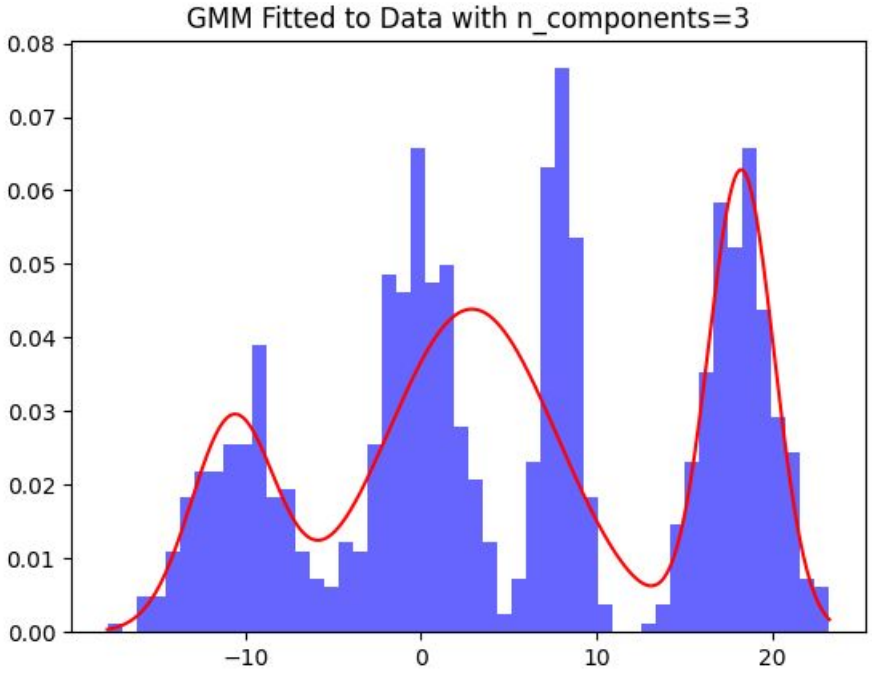
GMM: Using 4 Gaussians

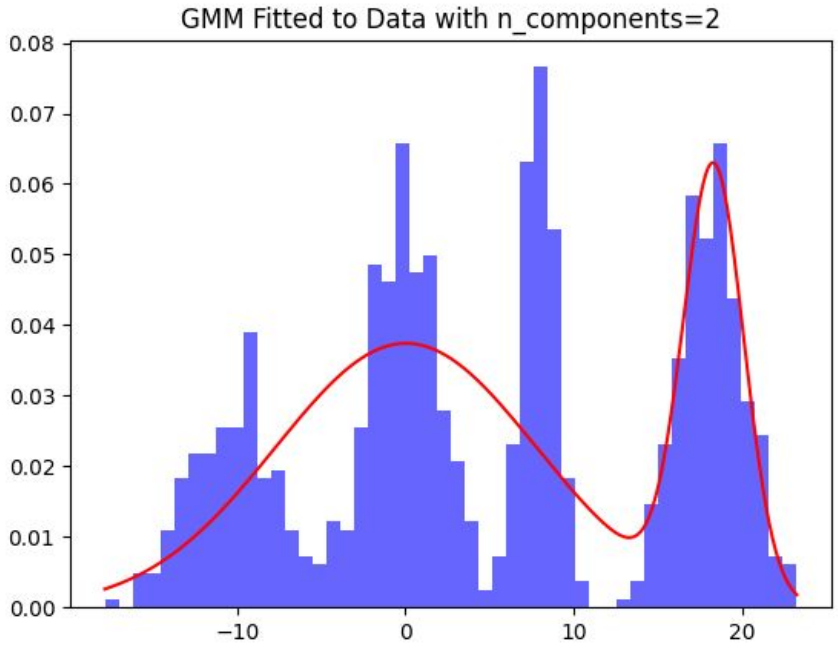
- If we asked GMM to find 4-gaussians, it will identify below gaussians



GMM: Less gaussians

- What if we used less gaussians? Below for 3 and 2





GMM Density Function

- GMM has interest math and details behind it. Study in the future
- GMM tries to model a probability **density** function p(x):
  - The mixture weight for each Gaussian component represents the probability that a **randomly selected data point** from the dataset belongs to that component

$$p(x) = \sum_{i=1}^{k} \pi_i \cdot \mathcal{N}(x|\mu_i, \Sigma_i)$$

where π_i is the mixture weight for the i^{th} Gaussian component, and $\mathcal{N}(x|\mu_i, \Sigma_i)$ is the Gaussian distribution with mean μ_i and covariance matrix Σ_i.

```
# Fit a Gaussian Mixture Model
n_components = 4
gmm = GaussianMixture(n_components=n_components, random_state=42)
gmm.fit(data)

# score_samples returns the log not the actual probability
# convert back the log to probability
log_likelihood = gmm.score_samples(x)
gmm_y = np.exp(log_likelihood)

probs = gmm.predict_proba(data)     # density for each sample: (1000, 4):
labels = gmm.predict(data)          # highest density

gmm.means_, gmm.covariances_, gmm.weights_
```

Log Likelihood log p(x)

$$\log p(x) = \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \right)$$

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2} \right)$$

```
def our_score_samples(gmm, x):
    def gaussian_density(x, mu, std):
        return (1 / (np.sqrt(2 * np.pi) * std)) * \
               np.exp(-0.5 * ((x - mu) / std) ** 2)

    sum_likelihood = np.zeros_like(x)
    zp = zip(gmm.means_.ravel(), gmm.covariances_.ravel(), gmm.weights_.ravel())
    for mu, variance, pi in zp:
        sum_likelihood += pi * gaussian_density(x, mu, std=np.sqrt(variance))

    return np.log(sum_likelihood).reshape(-1)

log_likelihood_our = our_score_samples(gmm, x)
```

Relevant Materials

- K-Means Assumptions
  - E.g. Assumes clusters to be **spherical** and equally sized, which is not always true.
- K-means objective function

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."