# CS 5785 Homework 3

Cornell Tech

Tian Jin[*†] Yufan Zhang[*‡]

October 2023

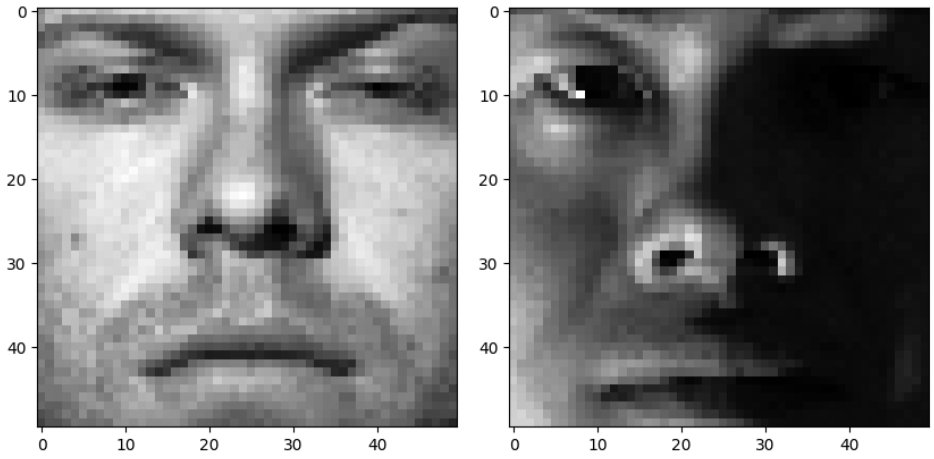
## 1 Programming Exercises

### 1.1 Eigenface for face recognition

#### Part (a) Download the dataset

#### Part (b) Load data and plot face

The code in Listing 1 is implemented to load the datasets and plot the sample faces from both the training and test datasets, which are shown in Figure 1.




(a) Traning dataset (b) Test dataset

Figure 1: Sample face images from the dataset

[*]Both authors contributed equally to this project.
[†]tj299@cornell.edu
[‡]yz2894@cornell.edu

```
# Loading training data
train_labels, train_data = [], []
for line in open('./faces/train.txt'):
  im = imageio.imread(line.strip().split()[0])
  train_data.append(im.reshape(2500,))
  train_labels.append(line.strip().split()[1])
train_data = np.array(train_data, dtype=float)
train_labels = np.array(train_labels, dtype=int)

print(train_data.shape, train_labels.shape)
plt.imshow(train_data[10, :].reshape(50,50), cmap = cm.Greys_r)
plt.show()

# Loading test data
test_labels, test_data = [], []
for line in open('./faces/test.txt'):
  im = imageio.imread(line.strip().split()[0])
  test_data.append(im.reshape(2500,))
  test_labels.append(line.strip().split()[1])
test_data = np.array(test_data, dtype=float)
test_labels = np.array(test_labels, dtype=int)

print(test_data.shape, test_labels.shape)
plt.imshow(test_data[10, :].reshape(50,50), cmap = cm.Greys_r)
plt.show()
```

Listing 1: Load the datasets and plot sample face

**Part (c) Average face**

The code in Listing 2 is implemented to get the average face $\mu$ from the whole training set, which is shown in Figure 2.

```
miu = train_data.sum(axis=0) / train_data.shape[0]
plt.imshow(miu.reshape(50,50), cmap = cm.Greys_r)
plt.show()
```

Listing 2: Getting the average face

**Part (d) Mean subtraction**

The code in Listing 3 is implemented to get the faces that were calculated by subtraction of the average face $\mu$. The resulting faces after mean subtraction for both the training and test datasets are shown in Figure 3.

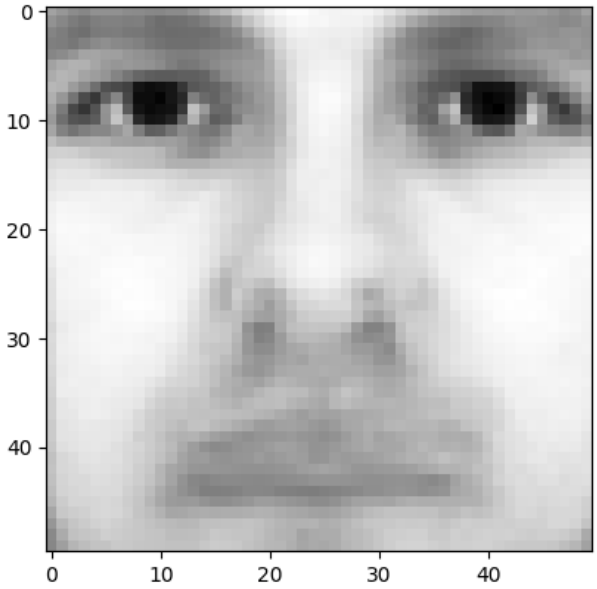



Figure 2: The calculated average face from the whole training set

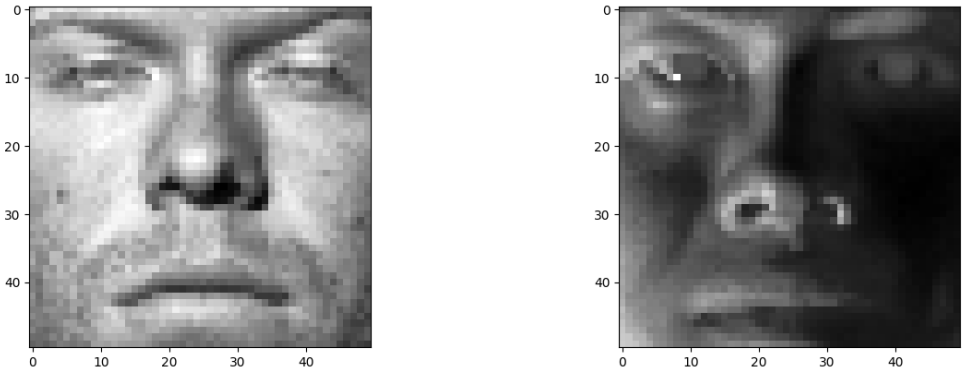
```
new_train = train_data - miu
plt.imshow(new_train[10, :].reshape(50,50), cmap = cm.Greys_r)
plt.savefig('img/mean_sub_train.png')
plt.show()

new_test = test_data - miu
plt.imshow(new_test[10, :].reshape(50,50), cmap = cm.Greys_r)
plt.savefig('img/mean_sub_test.png')
plt.show()
```

Listing 3: Getting the average face

**Part (e) Eigenface**

The code in Listing 4 is implemented to get the eigenfaces as instructed in the Homework description, whose results are visualized in Figure 4.



(a) Traning dataset (b) Test dataset

Figure 3: Face images after mean subtraction

(a) Image 9 (b) Image 10

Figure 4: The first ten eigenfaces in greyscale

```
def compute_V_T(X):
  # np.linalg.eig(): the column eigenvectors[:, i] is
  # the eigenvector corresponding to the eigenvalue eigenvalues[i]
  eigenvalues, eigenvectors = np.linalg.eig(np.dot(X.T, X))
  V_T = eigenvectors.T
  return V_T

train_V_T = compute_V_T(new_train)
for i in range(10):
  eigenface = np.real(train_V_T[i,:])
  plt.imshow(eigenface.reshape(50,50), cmap = cm.Greys_r)
  plt.savefig('img/eigenface_' + str(i) + '.png')
  plt.show()
```

Listing 4: Getting the eigenfaces

### Part (f) Eigenface feature

The code in Listing 5 and 6 are the functions defined to generate $r$-dimensional feature matrix for training images and test images, respectively.

```
train_V_T = compute_V_T(new_train)
def getTrainF(r):
  transpose = train_V_T[:r, :].T
  f = np.dot(train_data, transpose)
  return np.real(f)
```

Listing 5: Calculating Eigenface feature for training images

```
def getTestF(r):
  transpose = train_V_T[:r, :].T
  f = np.dot(test_data, transpose)
  return np.real(f)
```

Listing 6: Calculating Eigenface feature for test images

### Part (g) Face recognition

The code in Listing 7 is implemented for the logistic regression for face recognition, whose classification accuracy on the test set is **0.8**. The classification accuracy on the test set when $r = 1, 2, ...200$ is illustrated in Figure 5.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# extract training & test features
def extractFeatures(r):
  train_F = getTrainF(r)
  test_F = getTestF(r)
  return train_F, test_F

def logRegOvr(train_F, test_F):
  # train logistic regression model
  logRegOvr = LogisticRegression(
    multi_class="ovr", C=1e5,
    fit_intercept=True, solver='lbfgs', max_iter=1000
  )
  logRegOvr.fit(train_F, train_labels)
  # predict on test features
  pred_labels = logRegOvr.predict(test_F)
  # report accuracy
  acc = accuracy_score(test_labels, pred_labels)
  return ACC

train_F, test_F = extractFeatures(10)
acc = logRegOvr(train_F, test_F)
print("When r = 10, the classification accuracy on test set is", ACC)

# When r = 10, the classification accuracy on test set is 0.8

all_acc = []
for r in range(1, 201):
  train_F, test_F = extractFeatures(r)
  all_acc.append(logRegOvr(train_F, test_F))

plt.figure(figsize=(16,8))
plt.plot(range(1,201), all_acc, marker = 'o')
plt.grid(True)
plt.xlabel("Number of eigenfaces (r)")
plt.ylabel("Classification accuracy")
plt.show()
```

Listing 7: Logistic regression for face recognition

**Part (h) Low-Rank data loss**

The code in Listing 8 is to compute and plot the Frobenius distances over r = 1, 2, ..., 200, which is illustrated in Figure 6.

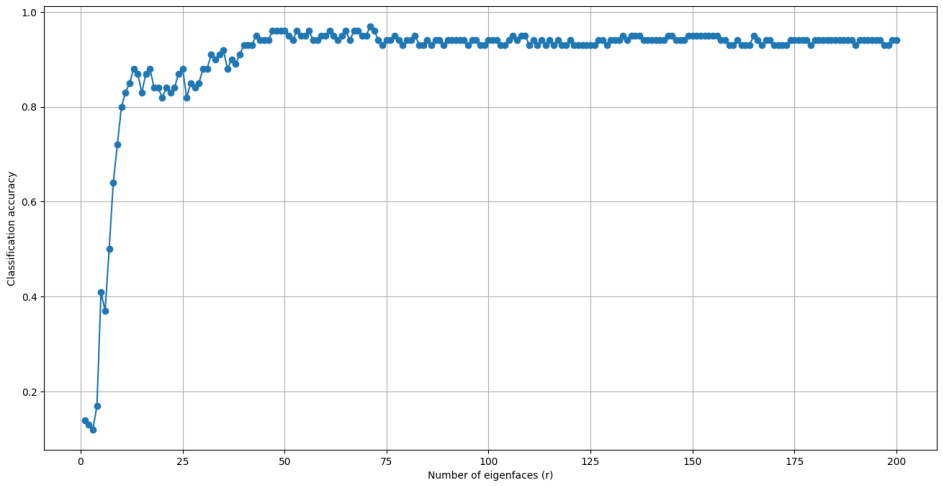



Figure 5: The classification accuracy on the test set when $r = 1, 2, ...200$

```
def compute_X_prime(X, V_T, r):
  # compute F given V_T & r:
  transpose = V_T[:r, :].T
  F = np.dot(X, transpose)
  # compute X_prime:
  X_prime = np.dot(F, V_T[:r, :])
  return X_prime

def compute_Frobenius(X, X_prime):
  diff = X - X_prime
  distance = np.sqrt(np.trace(np.dot(diff.T, diff)))
  return distance

all_dist = []
V_T = compute_V_T(train_data)

for r in range(1, 201):
  X_prime = compute_X_prime(train_data, V_T, r)
  dist = compute_Frobenius(train_data, X_prime)
  all_dist.append(dist)

plt.figure(figsize=(16,8))
plt.plot(range(1,201), all_dist, marker = 'o')
plt.grid(True)
plt.xlabel("Number of eigenfaces (r)")
plt.ylabel("Average Frobenius distance")
plt.show()
```

Listing 8: Compute and plot Frobenius distances over r = 1, 2, ... 200

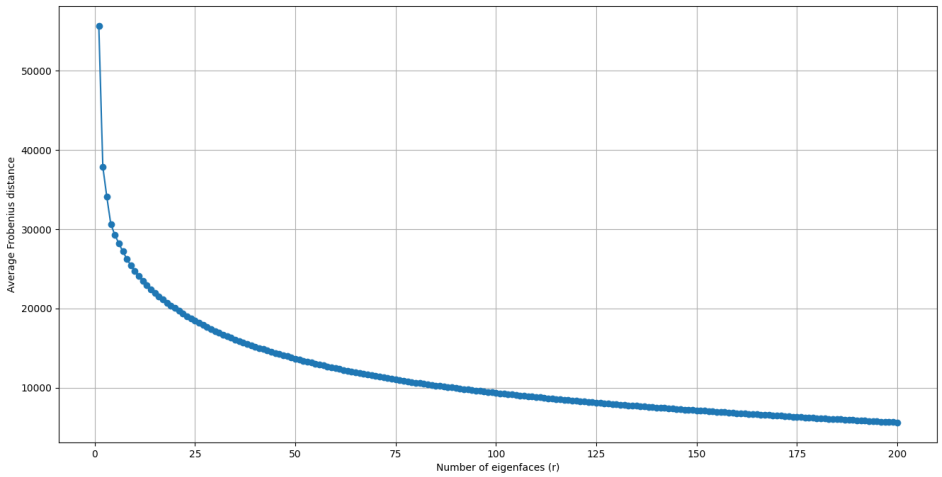



Figure 6: Frobenius distances over r = 1, 2, ... 200

## 1.2 Implement EM algorithm

**Part (a) Parse and plot all data points on 2-D plane.**

Listing 9 shows the script of parsing and plotting all data points on 2-D plane, which is shown in Figure 7.

```
with open('OldFaithfulGeyserData.txt') as f:
    data_array = [line.rstrip('\n') for line in f]

# remove all text explanations and read data points only
idx = data_array.index('    eruptions waiting')
data_array = data_array[idx+1:]

# parse string items to numeric items
data_lis = []
for str in data_array:
  entry_lis = list(map(float, str.split()[1:]))
  data_lis.append(entry_lis)

# convert python list to numpy array
data = np.array(data_lis)

# plot data points on 2d plane
plt.scatter(data[:, 0], data[:, 1], marker='o')
plt.xlabel('Eruption time in mins')
plt.ylabel('Waiting time to next eruption')
plt.grid(True)
plt.show()
```

Listing 9: Plot all data points on 2-d plane

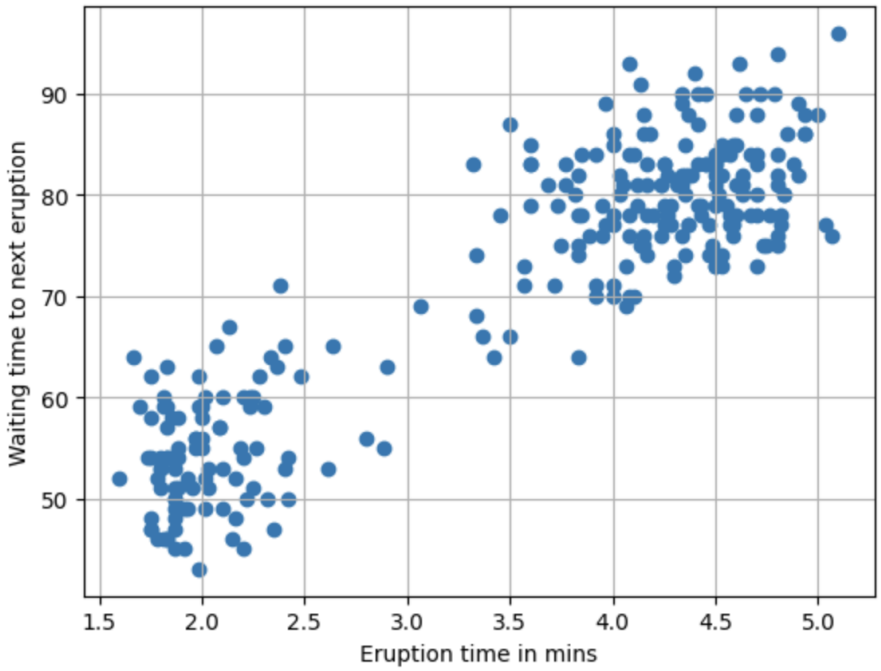



Figure 7: Plotted data points

**Part (b) Write down the expression for** $P_{\theta_t}(z = k|x)$

$$P_\theta(z = k|x) = \frac{P_\theta(z = k, x)}{P_\theta(x)} = \frac{P(x|z = k)P(z = k)}{\sum_{l=1}^{K} P_\theta(x|z = l)P(z = l)} = \frac{N(x; \mu_k, \Sigma_k) \cdot \phi_k}{\sum_{l=1}^{K} N(x; \mu_l, \Sigma_l) \cdot \phi_l}$$

**Part (c) Write down the formula for** $\mu_k, \Sigma_k, \phi$

$$\begin{aligned}\text{Goal}: \quad & \max_\theta \left( \sum_{k=1}^{K} \sum_{x \in D} P_{\theta_t}(z_k|x) \log P_\theta(x|z_k) + \sum_{k=1}^{K} \sum_{x \in D} P_{\theta_t}(z_k|x) \log P_\theta(z_k) \right) \\ & = \max_\theta \left( \sum_{k=1}^{K} \sum_{x \in D} P_{\theta_t}(z_k|x) \big( \log P_\theta(x|z_k) + \log P_\theta(z_k) \big) \right)\end{aligned}$$

Denoting $P_{\theta_t}(z = k|x)$ as $w_k$:

$$\text{Goal} \Leftrightarrow \max_\theta \left( \sum_{k=1}^{K} \sum_{x \in D} w_k \big( \log P_\theta(x|z_k) + \log P_\theta(z_k) \big) \right)$$

Taking the derivative of the objective w.r.t $\mu_k$ and setting it to zero, we get:

$$\mu_k^{t+1} = \frac{\sum_{x \in D} w_k x}{\sum_{x \in D} w_k}$$

Taking the derivative of the objective w.r.t $\Sigma_k$ and setting it to zero, we get:

$$\Sigma_k^{t+1} = \frac{\sum_{x \in D} w_k (x - \mu_k^{t+1})(x - \mu_k^{t+1})^T}{\sum_{x \in D} w_k}$$

Taking the derivative of the objective w.r.t $\phi_k$ and setting it to zero, we get:

$$\frac{\phi_k^{t+1}}{\sum_l \phi_l} = \frac{\sum_{x \in D} w_k}{n}$$

Considering the constraint that $\sum_l \phi_l = 1$, we get:

$$\phi_k^{t+1} = \frac{\sum_{x \in D} w_k}{n}$$

**Part (d) Implement and run the EM algorithm**

**i. Implementing EM algorithm from Scratch**

Listing 10 and 11 show the manual implementation of EM algorithm in Python. By running the function `implement_EM`, we can get the learned parameters for the given dataset.

```python
from scipy.stats import multivariate_normal

def initialize_parameters(data, num_clusters):
    num_data = data.shape[0]
    mu = data[np.random.choice(num_data, num_clusters)]
    sigma = num_clusters * [np.cov(data, rowvar=False)]
    phi = num_clusters * [1 / num_clusters]
    return mu, sigma, phi

def e_step(data, num_clusters, mu, sigma, phi):
    # initialize parameters
    num_data = data.shape[0]
    posterior = np.zeros((num_data, num_clusters))
    # compute parameters
    for k in range(num_clusters):
        posterior[:, k] = phi[k] * multivariate_normal.pdf(
            data, mean=mu[k], cov=sigma[k]
        )
    return posterior / posterior.sum(axis=1, keepdims=True)

def m_step(data, num_clusters, posterior):
    # initialize parameters
    num_data = data.shape[0]
    num_features = data.shape[1]
    mu = np.zeros((num_clusters, num_features))
    sigma = np.zeros((num_clusters, num_features, num_features))
    phi = np.zeros(num_clusters)
    # compute parameters
    for k in range(num_clusters):
        phi_k = posterior[:, k].sum()
        mu[k] = (posterior[:, k].reshape(-1, 1) * data).sum(axis=0) / phi_k
        numerator = np.dot(
            (posterior[:, k].reshape(-1, 1) * (data - mu[k])).T, (data - mu[k])
        )
        sigma[k] = numerator / phi_k
        phi[k] = phi_k / num_data
    return mu, sigma, phi

def objective(data, num_clusters, mu, sigma, phi):
    likelihood = np.zeros((data.shape[0], num_clusters))
    for k in range(num_clusters):
        likelihood[:, k] = phi[k] * multivariate_normal.pdf(
            data, mean=mu[k], cov=sigma[k]
        )
    return np.sum(np.log(likelihood.sum(axis=1)))
```

Listing 10: The E-Step and M-Step Implementation of the EM algorithm

```python
def implement_EM(data, num_clusters=2, num_iterations=100, threshold=1e-6):
    mu, sigma, phi = initialize_parameters(data, num_clusters)
    log_likelihood_prev = 0
    for k in range(num_iterations):
        # e-step
        posterior = e_step(data, num_clusters, mu, sigma, phi)
        # m-step
        mu, sigma, phi = m_step(data, num_clusters, posterior)
        # compute log-likelihood
        log_likelihood_current = objective(data, num_clusters, mu, sigma, phi)
        # check for convergence
        if abs(log_likelihood_current - log_likelihood_prev) < threshold:
            print(f"Converged at iteration {k}")
            break
        # update log-likelihood
        log_likelihood_prev = log_likelihood_current
    return mu, sigma, phi

mu, sigma, phi = implement_EM(data, num_clusters=2)
print(f"mu: {mu}\n")
print(f"sigma: {sigma}\n")
print(f"phi: {phi}\n")

# Converged at iteration 9
# mu: [[ 4.28966404 79.96814021] [ 2.03639079 54.47853991]]
# sigma: [[[ 0.16996581 0.94057589] [ 0.94057589 36.04583496]]
# [[ 0.06916953 0.43518701] [ 0.43518701 33.69741426]]]
# phi: [0.64412618 0.35587382]
```

Listing 11: Running the implemented EM algorithm

### ii. Explanation of the termination criterion

The termination criterion is based on the change in the log-likelihood between consecutive iterations of the EM algorithm. The reasoning is as follows:

1. **Monotonicity of the Log-likelihood**: The log-likelihood should never decrease in the EM algorithm. By monitoring the change in log-likelihood, we can assess whether the algorithm is still making meaningful updates to the parameters.

2. **Early Stopping**: If the improvement in the log-likelihood is smaller than the threshold (`1e-6` in this case), the algorithm terminates, making the algorithm more efficient by avoiding unnecessary iterations.

### iii. Trajectories of the two mean vectors

Before plotting the trajectories of the two mean vectors ($\mu_1$ and $\mu_2$), the modified `implement_EM` in Listing 12 to adopted to store the means during the training process.

```python
# Modify the implement_EM function to store the posterior probabilities
def implement_EM(data, num_clusters=2, num_iterations=100, threshold=1e-6):
    mu, sigma, phi = initialize_parameters(data, num_clusters)
    log_likelihood_prev = -np.inf
    mu_trajectory = {k: [] for k in range(num_clusters)}
    post_prob = []  # Store posterior probabilities

    for k in range(num_iterations):
        # Store the means
        for i in range(num_clusters):
            mu_trajectory[i].append(mu[i].copy())
        # e-step
        posterior = e_step(data, num_clusters, mu, sigma, phi)
        post_prob.append(posterior.copy())  # Store posterior
        # m-step
        mu, sigma, phi = m_step(data, num_clusters, posterior)
        # compute log-likelihood
        log_likelihood_current = objective(data, num_clusters, mu, sigma, phi)
        # check for convergence
        if abs(log_likelihood_current - log_likelihood_prev) < threshold:
            break
        log_likelihood_prev = log_likelihood_current

    return mu, sigma, phi, mu_trajectory, post_prob  # Return posterior

# Run the EM algorithm and get the trajectories and posteriors
mu, sigma, phi, mu_trajectory, post_prob = implement_EM(data, num_clusters=2)
```

Listing 12: Modified EM implementation for storing the means

To plot the trajectories of mean vectors, the code in Listing 13 is implemented. Additionally, we plotted the clustered data points in the figure as well to illustrate the final results of the EM algorithm, as shown in Figure 8.

```python
# Assign colors to data points based on highest posterior probability
colors = ["red", "blue"]

# Create a scatter plot for clustered data points
last_posterior = post_prob[-1]  # Using posteriors from the last iteration
cluster_assignments = np.argmax(last_posterior, axis=1)

plt.figure(figsize=(10, 6))
for i in range(2):  # Assuming two clusters
    cluster_data = data[cluster_assignments == i]
    plt.scatter(cluster_data[:, 0], cluster_data[:, 1], color=colors[i],
        alpha=0.2, label=f"Cluster {i+1} data")
    # Plotting the trajectories of the means
    mu_x, mu_y = zip(*mu_trajectory[i])
    plt.plot(mu_x, mu_y, color=colors[i], marker="o",
        linestyle="-", linewidth=2, markersize=5,
        label=f"Mean trajectory of cluster {i+1}")

plt.xlabel("X-coordinate")
plt.ylabel("Y-coordinate")
plt.title("Clustered Data Points and Trajectories of Mean Vectors")
plt.legend()
plt.grid()
plt.show()
```

Listing 13: Plotting the clustered data points and trajectories of mean vectors

**Part (e) Experiment with K-means clustering and comment**

Given the same dataset, we implemented a K-means clustering to examine its difference with the GMM model fit using the EM algorithm, as implemented in List 14. The clustering result is visualized in Figure 9. Upon analyzing the clustering results, there are discernible differences between the clusters formed by K-means and the EM algorithm (GMM). As shown in Figure 9, K-means seems to prioritize the data's vertical (Y-axis) spread, resulting in a horizontal boundary between the clusters. In contrast, as Figure 8 shows, the GMM seems to capture more nuanced clusters with elliptical shapes, which is evident from the trajectories of the mean vectors. The K-means algorithm partitions the data in a more definite manner by assigning each to the closest cluster following Euclidean distance, while EM-GMM provides a probabilistic approach, which can result in softer boundaries and more overlap between clusters.

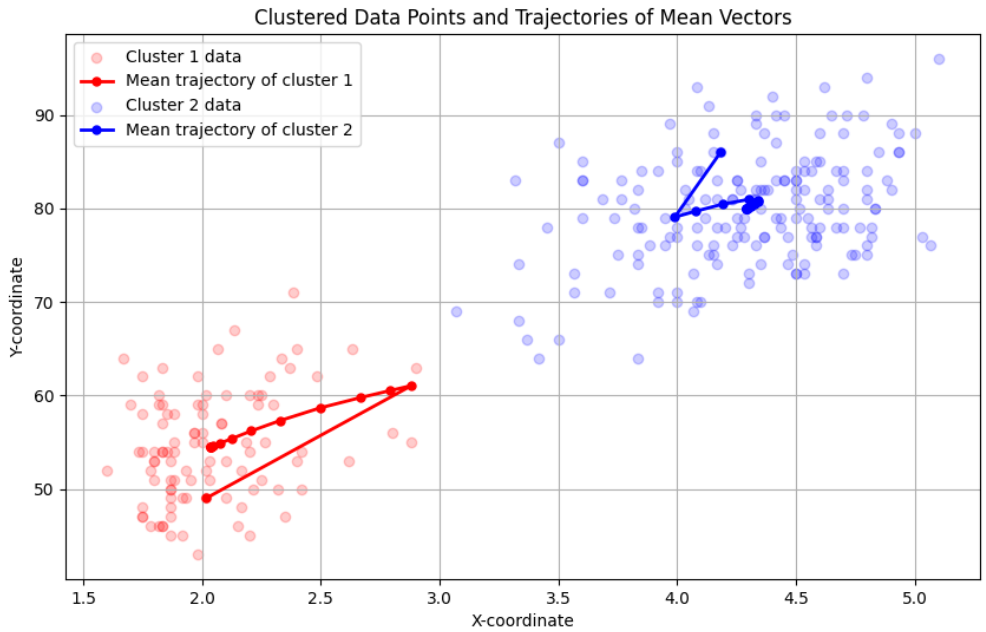



Figure 8: Clustered data points and trajectories of mean vectors

```python
from sklearn.cluster import KMeans

# k-means clustering
kmeans = KMeans(n_clusters=2,andom_state=42)
labels = kmeans.fit_predict(data)

# Plot
plt.figure(figsize=(10, 6))
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap="viridis", marker="o")
plt.scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
    s=200, c="red", marker="o",
)
plt.title("K-means Clustering")
plt.xlabel("Eruption time in mins")
plt.ylabel("Waiting time to next eruption")
plt.grid(True)
plt.savefig("img/kmeans.png")
plt.show()
```

Listing 14: K-means clustering on the same dataset

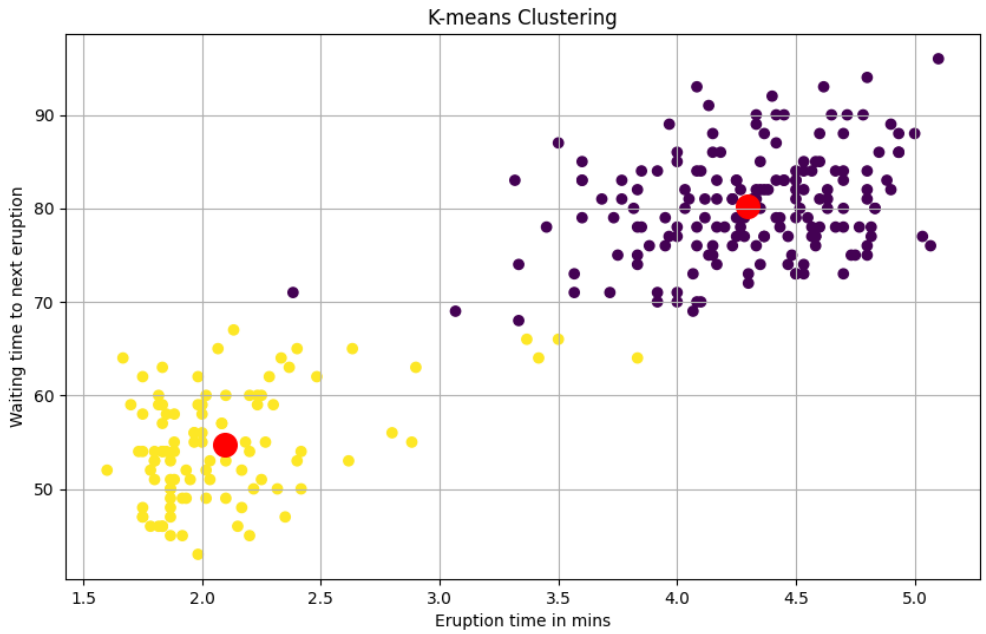



Figure 9: Clustered data points with K-means

## 1.3 Kernel Density Estimation

**Part (a) Plot the density estimate**

We are given the following data:

$$26, 30, 27, 18, 75, 66, 73, 63, 56, 83$$

Before implementing and plotting the kernel density estimate (KDE) of the given dataset, we need to first use the code in Listing 15 to prepare our data and the range of $x$ values for plotting.

```
# Given data
data = np.array([26, 30, 27, 18, 75, 66, 73, 63, 56, 83]).reshape(-1, 1)

# Values of x for the density estimate
x_vals = np.linspace(min(data)-20, max(data)+20, 1000).reshape(-1, 1)
```

Listing 15: Prepare the data, bandwidths, and x value for plotting

Listing 16 shows the code to plot the KDE of the given dataset using a Gaussian kernel with Sklearn.

```python
from sklearn.neighbors import KernelDensity

# Kernel density estimation for  = 100
kde_100 = KernelDensity(kernel='gaussian', bandwidth=100).fit(data)
log_density_100 = kde_100.score_samples(x_vals)

# Kernel density estimation for  = 10
kde_10 = KernelDensity(kernel='gaussian', bandwidth=10).fit(data)
log_density_10 = kde_10.score_samples(x_vals)

# Plotting the density estimates
plt.figure(figsize=(10, 6))
plt.plot(x_vals, np.exp(log_density_100), label='$\delta=100$')
plt.plot(x_vals, np.exp(log_density_10), label='$\delta=10$')
plt.scatter(data, np.zeros_like(data), color='red', label='Data points')
plt.title('Kernel Density Estimate with Gaussian Kernel (Using Sklearn)')
plt.legend()
plt.xlabel('x')
plt.ylabel('Density')
plt.grid(True)
plt.savefig(IMG_PATH + 'sklearn_kde.png')
plt.show()
```

Listing 16: Plot the kernel density estimate (KDE) using a Gaussian kernel with Sklearn

Moreover, we manually implement the KDE using NumPy and plot the density estimates. The formula for the KDE with a Gaussian kernel is:

$$\tilde{p}(x) = \frac{1}{n\delta} \sum_{i=1}^{n} K\left(\frac{x - x_i}{\delta}\right)$$

where

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$$

Listing 16 shows the code to plot the KDE of the given dataset using a Gaussian kernel, manually calculated with bandwidth parameters $\delta = 100$ and $\delta = 10$.

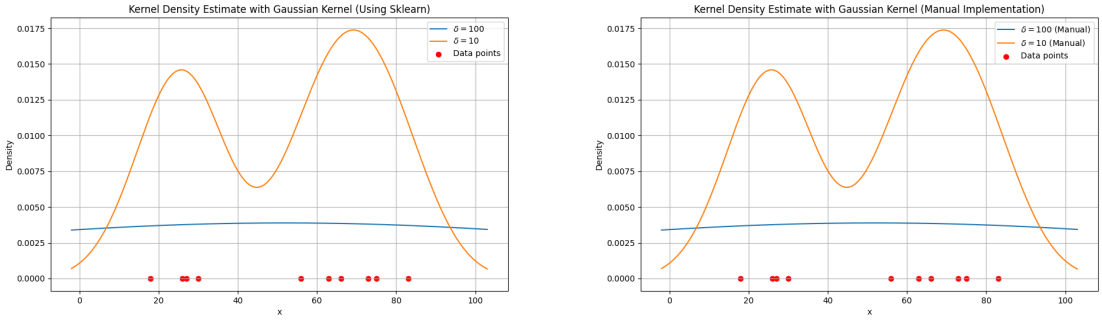
Figure 10 illustrates the kernel density estimate of the given dataset using a Gaussian kernel, as computed by the `sklearn` library and manual implementation with bandwidth parameters $\delta = 100$ and $\delta = 10$. The results are consistent between these two implementations.

```python
# Function to calculate Gaussian kernel density estimate
def gaussian_kde(x, data, bandwidth):
    n = len(data)
    constant = 1 / (np.sqrt(2 * np.pi) * bandwidth)
    densities = np.sum(
        [constant * np.exp(-((x - xi) ** 2) / (2 * bandwidth**2)) for xi in data],
        axis=0,
    )
    return densities / n


# Calculating densities manually
density_100_manual = gaussian_kde(x_vals, data, bandwidth=100)
density_10_manual = gaussian_kde(x_vals, data, bandwidth=10)

# Plotting the manually calculated densities
plt.figure(figsize=(10, 6))
plt.plot(x_vals, density_100_manual, label="$\delta=100$ (Manual)")
plt.plot(x_vals, density_10_manual, label="$\delta=10$ (Manual)")
plt.scatter(data, np.zeros_like(data), color="red", label="Data points")
plt.title("Kernel Density Estimate with Gaussian Kernel (Manual Implementation)")
plt.xlabel("x")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.savefig(IMG_PATH + "manual_kde.png")
plt.show()
```

Listing 17: Plot the kernel density estimate (KDE) using a Gaussian kernel with manual implementation




(a) Computed by the `sklearn` library

(b) Computed via manual implementation

Figure 10: KDE with bandwidth parameters $\delta = 100$ and $\delta = 10$

**Part (b) Compare the two bandwidths**

The choice of $\delta = 10$ is a preferable choice for this dataset, as justified in the following reasons:

- Bandwidth $\delta = 10$: The curve is more wiggly and appears to fit closely to the data. It captures the individual peaks in the data distribution, which may indicate local variations in the data.
- Bandwidth $\delta = 100$: This curve is smoother and provides a broad overview of the data. It essentially averages out the local variations and might be oversimplifying the data distribution.

Method for choosing $\delta$:

- **Cross-Validation**: We can partition the data into training and validation sets, compute the KDE for the training set using a given bandwidth, and then evaluate how well this KDE predicts the validation set. The bandwidth that performs best (typically measured by log-likelihood) on the validation set is chosen.
- **Visual Inspection**: As demonstrated here, a visual inspection of the KDE plot for different bandwidths can give a good idea of which bandwidth provides a better representation of the data.

**Part (c) Probability of new points**

The code in Listing 18 is used to compute the density $\tilde{p}(x)$ at the new samples $x = 30$ and $x = 95$ for both $\delta = 100$ and $\delta = 10$.

```
kde_100 = KernelDensity(kernel='gaussian', bandwidth=100).fit(data)
kde_10 = KernelDensity(kernel='gaussian', bandwidth=10).fit(data)

# New samples
new_samples = np.array([30, 95]).reshape(-1, 1)

# Calculating the density estimates for the new points
density_new_100 = kde_100.score_samples(new_samples)
density_new_10 = kde_10.score_samples(new_samples)

# Returning the density values of the new points
print(np.exp(density_new_100), np.exp(density_new_10))

# [0.00380071 0.00355839] [0.01358759 0.00292187]
```

Listing 18: Calculate the probability of two new samples $x = 30$ and $x = 95$

The densities $\tilde{p}(x)$ at the new samples $x = 30$ and $x = 95$ were calculated using `sklearn` implementation. The results are as follows:

- For $\delta = 100$:

$$\tilde{p}(30) \approx 0.003801, \tilde{p}(95) \approx 0.003558$$

- For $\delta = 10$:

$$\tilde{p}(30) \approx 0.013588, \tilde{p}(95) \approx 0.002922$$

Furthermore, we use the code in Listing 19 to visualize these new points on the previous plots, as shown in Figure 11. The KDE are displayed with the new samples $x = 30$ and $x = 95$ highlighted as green dots for bandwidth parameters $\delta = 100$ and $\delta = 10$.

```
# Plotting the density estimates with new points
plt.figure(figsize=(10, 6))
plt.plot(x_vals, np.exp(log_density_100), label="$\delta=100$")
plt.plot(x_vals, np.exp(log_density_10), label="$\delta=10$")
plt.scatter(data, np.zeros_like(data), color="red", label="Data points")
plt.scatter(
    new_samples, np.zeros_like(new_samples), color="blue",
    label="New points"
)

# Highlighting new samples
plt.scatter(
    new_samples, np.exp(density_new_100), color="green",
    label="New points (Density)"
)
plt.scatter(
    new_samples, np.exp(density_new_10), color="green",
    label="New points (Density)"
)

plt.title("Kernel Density Estimate with New Points")
plt.xlabel("x")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.savefig(IMG_PATH + "new_points.png")
plt.show()
```

Listing 19: Visualize the new samples in the previous KDE plot

**Part (d) Rule to accept or reject points**

A possible rule to determine whether a point is an outlier based on KDE is by setting a threshold density value, $\tau$. A point $x$ is considered part of the data distribution if its density $\tilde{p}(x)$ is above the threshold:

$$\tilde{p}(x) > \tau$$

Otherwise, it is considered an outlier.

However, this strategy might not always work effectively, especially when the data distribution is multi-

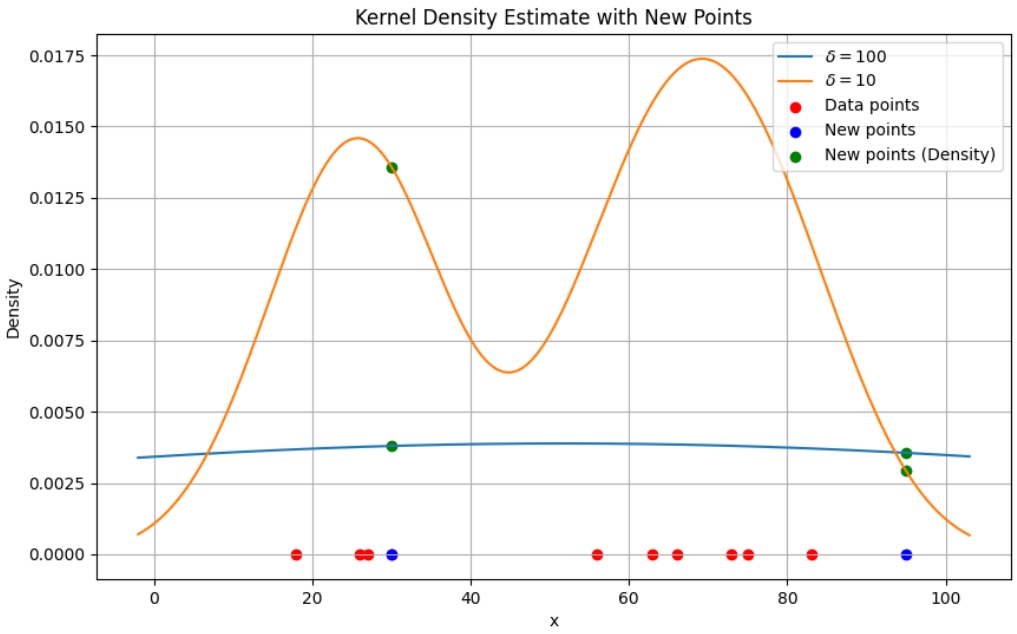



Figure 11: KDE with new samples highlighted

modal or has varying densities in different regions. In such cases, a global threshold might not suitably differentiate between typical points and outliers. Additionally, the choice of kernel and bandwidth also influences the KDE, making the rule sensitive to these hyperparameters.

**Part (e) Compare Gaussian and Tophat kernels**

The code in Listing 20 change the kernel to a Tophat kernel and plot the KDE for $\delta = 10$. The Tophat kernel is defined as:

$$K(x, z; \delta) = \begin{cases} 1 & \text{if } \|x - z\| \leq \frac{\delta}{2} \\ 0 & \text{otherwise} \end{cases}$$

Figure 12 visualizes the kernel density estimate (KDE) using the Tophat kernel with a bandwidth parameter $\delta = 10$.

One shortcoming of using the Tophat kernel for outlier detection compared to the Gaussian kernel is its abrupt transition at the boundaries of the kernel window. This can lead to a less smooth and less generalized estimate of the data distribution, making it harder to differentiate between typical points and outliers effectively.

Let's consider a one-dimensional dataset mostly distributed between values $10$ and $20$ with a single outlier at value $35$.

- **Tophat Kernel**: For a Tophat kernel with a bandwidth parameter, $\delta = 20$. the outlier might not be effectively identified as the KDE might be influenced heavily due to the uniform weighting within the kernel window.
- **Gaussian Kernel**: For a Gaussian kernel, even if the kernel window includes the outlier, it would

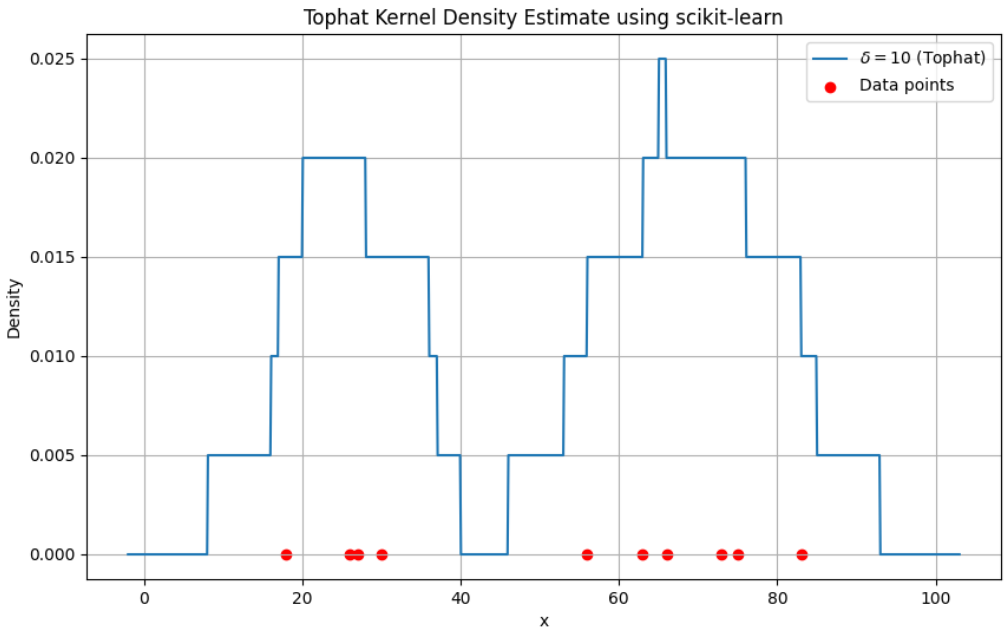



Figure 12: KDE using Tophat kernal with $\delta = 10$

assign a lower weight to it due to the decreasing weight assigned to points as they move away from the center of the kernel. In other words, the Gaussian kernel, with its smooth weight transition, can more effectively identify the outlier by assigning lower weights to distant points.

```python
# Kernel density estimation with Tophat kernel for  = 10
kde_tophat_10 = KernelDensity(kernel="tophat", bandwidth=10).fit(
    data.reshape(-1, 1)
)
log_density_tophat_10 = kde_tophat_10.score_samples(x_vals)

# Plotting the Tophat kernel density estimates
plt.figure(figsize=(10, 6))
plt.plot(x_vals, np.exp(log_density_tophat_10), label="$\delta=10$ (Tophat)")
plt.scatter(data, np.zeros_like(data), color="red", label="Data points")
plt.title("Tophat Kernel Density Estimate using scikit-learn")
plt.xlabel("x")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.savefig(IMG_PATH + "tophat_kde.png")
plt.show()
```

Listing 20: Plot the Tophat density estimate of the dataset using $\delta = 10$

# 2 Written Exercises

## 2.1 K-means clustering

**Part (a)** $J(c_{t-1}, f_t) \leq J(c_{t-1}, f_{t-1})$

*Proof.* Consider a data point $x^{(i)}$. Let $c_{t-1}^{(a)}$ be the closest centroid to $x^{(i)}$ at time $t-1$, and let $c_{t-1}^{(b)}$ be the centroid assigned to $x^{(i)}$ at time $t-1$. Thus, we have:

$$||x^{(i)} - c_{t-1}^{(a)}||_2 \leq ||x^{(i)} - c_{t-1}^{(b)}||_2 \tag{1}$$

Squaring both sides to work with squared distances, and summing over all data points, we get:

$$\sum_{i=1}^{n} ||x^{(i)} - c_{t-1}^{(a)}||_2^2 \leq \sum_{i=1}^{n} ||x^{(i)} - c_{t-1}^{(b)}||_2^2 \tag{2}$$

After Step 1 of the K-means iteration, $c_{t-1}^{(a)}$ becomes the centroid for the new assignment of $x^{(i)}$, i.e., $c_{t-1}^{f_t(x^{(i)})}$. Using the notation $c_{t-1}^{f_t(x^{(i)})}$ and $c_{t-1}^{f_{t-1}(x^{(i)})}$ to represent the centroids, we rewrite the equation as:

$$\sum_{i=1}^{n} ||x^{(i)} - c_{t-1}^{f_t(x^{(i)})}||_2^2 \leq \sum_{i=1}^{n} ||x^{(i)} - c_{t-1}^{f_{t-1}(x^{(i)})}||_2^2 \tag{3}$$

This simplifies to:

$$J(c_{t-1}, f_t) \leq J(c_{t-1}, f_{t-1}) \tag{4}$$

Concluding, the objective function does not increase and therefore, the K-means algorithm is monotonically decreasing. □

**Part (b):** $J(c_t, f_t) \leq J(c_{t-1}, f_t)$

*Proof.* Consider a single cluster, denoted by the index $k$, and let the points in this cluster be given by $\{x^{(i)} : f_t(x^{(i)}) = k\}$. The objective is to show that the updated centroid, $c_t^{(k)}$, which is the mean of the points in cluster $k$, minimizes the sum of squared Euclidean distances to the points in the cluster. Let us denote this sum as $M_k(c)$, which is formulated as follows:

$$M_k(c) = \sum_{i:f_t(x^{(i)})=k} ||x^{(i)} - c||_2^2$$

To find the value of $c$ that minimizes $M_k(c)$, we take the derivative with respect to $c$ and set it to zero:

$$\frac{dM_k(c)}{dc} = -2 \sum_{i:f_t(x^{(i)})=k} (x^{(i)} - c)$$

Solving the equation $\frac{dM_k(c)}{dc} = 0$, we get:

$$\sum_{i:f_t(x^{(i)})=k} (x^{(i)} - c) = 0$$

$$\sum_{i:f_t(x^{(i)})=k} x^{(i)} = cS^{(k)}$$

$$c = \frac{1}{S^{(k)}} \sum_{i:f_t(x^{(i)})=k} x^{(i)}$$

$$c = c_t^{(k)}$$

This confirms that the value of $c$ which minimizes $M_k(c)$ is indeed the updated centroid $c_t^{(k)}$. Consequently, updating the centroids in this manner also ensures that the objective function $J(c_t, f_t)$ is minimized, since it is the sum of Euclidean distances, establishing that:

$$J(c_t, f_t) \leq J(c_{t-1}, f_t)$$

□

## 2.2 Local Optima in K-means

Consider a dataset with the following 1D points:

$$[0, 2, 6, 11, 12, 13]$$

We want to partition these points into two clusters ($K = 2$).

**Initialization 1:** Centroids: $C_1 : 2$, $C_2 : 12$

Iteration 1: $\begin{cases} \text{Points closer to } C_1 : [0, 2, 6] \\ \text{Points closer to } C_2 : [11, 12, 13] \end{cases}$

New centroids: $C_1 : \frac{0+2+6}{3} = 2.67$, $C_2 : \frac{11+12+13}{3} = 12$

Iteration 2: $\begin{cases} \text{Points closer to } C_1 : [0, 2, 6] \\ \text{Points closer to } C_2 : [11, 12, 13] \end{cases}$

New centroids: $C_1 : \frac{0+2+6}{3} = 2.67$, $C_2 : \frac{11+12+13}{3} = 12$

Stop the iteration here because the centroids do not change anymore.

**Initialization 2:** Centroids: $C_1 : 0$, $C_2 : 6$

Iteration 1: $\begin{cases} \text{Points closer to } C_1 : [0, 2] \\ \text{Points closer to } C_2 : [6, 11, 12, 13] \end{cases}$

New centroids: $C_1 : \dfrac{0+2}{2} = 1$, $C_2 : \dfrac{6+11+12+13}{4} = 10.5$

Iteration 2: $\begin{cases} \text{Points closer to } C_1 : [0, 2] \\ \text{Points closer to } C_2 : [6, 11, 12, 13] \end{cases}$

New centroids: $C_1 : \dfrac{0+2}{2} = 1$, $C_2 : \dfrac{6+11+12+13}{4} = 10.5$

Stop the iteration here because the centroids do not change anymore.

In **Initialization 1**, the final clusters are $[0, 2, 6]$ and $[11, 12, 13]$. In **Initialization 2**, the final clusters are $[0, 2]$ and $[6, 11, 12, 13]$, showcasing the sensitivity of K-means to initial centroid choices.