Hw 2A || CS 6220 || Nickhil Tekwani

Problem 1: K Means Theory

A) E-Step Update

In the E-step, we fix the centroids μ and assign each data point X_i to the nearest centroid.

For a given centroid μ_k, if it's the closest centroid to the point X_i, then pi_ik = 1. Otherwise, pi_ik = 0.

The objective function given the centroids is:

$$J = \sum_i \sum_k \pi_{ik} ||X_i - \mu_k||^2$$

Given μ_k, to minimize (J) for each (X_i), we should set pi_ik to 1 for the closest centroid and 0 for all others. Thus, the E-step achieves the minimum objective for the current centroids.

B) M-Step Update

In the M-step, given assignments pi, we need to update the centroids μ to minimize the objective. For each cluster (k), the centroid is updated as the mean of all the data points assigned to that cluster:

$$\mu_k = \frac{\sum_i \pi_{ik} X_i}{\sum_i \pi_{ik}}$$

To prove that this choice of μ_k minimizes the objective function, consider differentiating the objective with respect to μ_k and setting it to zero.

$$\frac{\partial J}{\partial \mu_k} = 2 \sum_i \pi_{ik} (\mu_k - X_i)$$

Setting the above to zero and solving for μ_k gives:

$$\mu_k = \frac{\sum_i \pi_{ik} X_i}{\sum_i \pi_{ik}}$$

Thus, the M-step update for μ_k minimizes the objective for the given memberships pi.

C) Convergence of KMeans

KMeans is guaranteed to converge because the objective function is monotonically non-increasing. In each iteration (combining the E-step and M-step), the objective either decreases or remains the same.

However, KMeans might not reach the global minimum of the objective function because the algorithm can get stuck in local minima. This behavior is influenced by the initial choice of centroids. Different runs of KMeans with different initializations can lead to different final cluster assignments and centroids.

Problem 2: K Means on Data

In [1]:
```
import numpy as np

class KMeans:
    def __init__(self, k, distance="euclidean", max_iters=100, tol=1e-4):
        self.k = k
        self.max_iters = max_iters
        self.tol = tol
        self.centroids = None
        self.distance = distance

    def _calculate_distance(self, x1, x2):
        if self.distance == "euclidean":
            return np.linalg.norm(x1 - x2)
        elif self.distance == "dot":
            return -np.dot(x1, x2)
        else:
            raise ValueError("Unknown distance metric!")

    def fit(self, X):
        # Initialize centroids randomly from the data points
        self.centroids = X[np.random.choice(X.shape[0], self.k, replace=Fal
        prev_centroids = np.zeros_like(self.centroids)

        for _ in range(self.max_iters):
            distances = np.array([[self._calculate_distance(x, centroid) fo
            clusters = np.argmin(distances, axis=1)

            # Update Centroids
            new_centroids = np.array([X[clusters == i].mean(axis=0) for i i

            # Check for convergence
            if np.linalg.norm(new_centroids - prev_centroids) < self.tol:
                break

            prev_centroids = new_centroids
            self.centroids = new_centroids

    def predict(self, X):
        distances = np.array([[self._calculate_distance(x, centroid) for ce
        return np.argmin(distances, axis=1)
```

In [13]:

```
from sklearn.metrics import confusion_matrix

def kmeans_objective(X, labels, centroids):
    return sum(np.linalg.norm(X[i] - centroids[labels[i]])**2 for i in rang

def purity_score(y_true, y_pred):
    # Convert both sets of labels to string type
    y_true_str = [str(label) for label in y_true]
    y_pred_str = [str(label) for label in y_pred]

    # Compute confusion matrix
    matrix = confusion_matrix(y_true_str, y_pred_str)

    # Return purity
    return np.sum(np.amax(matrix, axis=0)) / np.sum(matrix)

def gini_index(y_true, y_pred):
    # Convert both sets of labels to string type
    y_true_str = [str(label) for label in y_true]
    y_pred_str = [str(label) for label in y_pred]

    matrix = confusion_matrix(y_true_str, y_pred_str)
    total_samples = np.sum(matrix)
    sum_gini = 0

    # Iterate over each cluster
    for cluster in matrix.T:  # Transpose to get clusters
        # Proportions of each class in the cluster
        proportions = cluster / np.sum(cluster)
        gini = 1 - np.sum(proportions ** 2)

        # Weighted Gini (by the size of the cluster)
        sum_gini += gini * np.sum(cluster)

    return sum_gini / total_samples
```

In [7]:
```
from sklearn.decomposition import TruncatedSVD

def load_data(name, n_samples=5000, n_features=30):
    if name == "MNIST":
        data = fetch_openml('mnist_784', cache=True)
        X, y = data["data"].values, data["target"].values
    elif name == "FASHION":
        data = fetch_openml('Fashion-MNIST', cache=True)
        X, y = data["data"].values, data["target"].values
    elif name == "20NG":
        newsgroups = fetch_20newsgroups(subset='all', remove=('headers', 'f
        vectorizer = TfidfVectorizer(max_features=2000, stop_words='english
        X = vectorizer.fit_transform(newsgroups.data)
        y = newsgroups.target
    else:
        raise ValueError("Unknown dataset!")

    if name == "20NG":
        reducer = TruncatedSVD(n_components=n_features)
    else:
        reducer = PCA(n_components=n_features)

    X_reduced = reducer.fit_transform(X[:n_samples])

    return X_reduced, y[:n_samples]

mnist_data, mnist_labels = load_data("MNIST")
fashion_data, fashion_labels = load_data("FASHION")
ng_data, ng_labels = load_data("20NG")
```

```
/usr/local/lib/python3.10/site-packages/sklearn/datasets/_openml.py:968:
FutureWarning: The default value of `parser` will change from `'liac-arf
f'` to `'auto'` in 1.4. You can set `parser='auto'` to silence this warni
ng. Therefore, an `ImportError` will be raised from 1.4 if the dataset is
dense and pandas is not installed. Note that the pandas parser may return
different data types. See the Notes Section in fetch_openml's API doc for
details.
  warn(
/usr/local/lib/python3.10/site-packages/sklearn/datasets/_openml.py:968:
FutureWarning: The default value of `parser` will change from `'liac-arf
f'` to `'auto'` in 1.4. You can set `parser='auto'` to silence this warni
ng. Therefore, an `ImportError` will be raised from 1.4 if the dataset is
dense and pandas is not installed. Note that the pandas parser may return
different data types. See the Notes Section in fetch_openml's API doc for
details.
  warn(
```

In [14]:

```
datasets = {
    "MNIST": (mnist_data, mnist_labels, 10),
    "FASHION": (fashion_data, fashion_labels, 10),
    "20NG": (ng_data, ng_labels, 20)
}

results = {}

for name, (data, labels, k) in datasets.items():
    km = KMeans(k=k)
    km.fit(data)
    preds = km.predict(data)

    results[name] = {
        "Objective": kmeans_objective(data, preds, km.centroids),
        "Purity": purity_score(labels, preds),
        "Gini Index": gini_index(labels, preds)
    }

    # Additional runs for higher and lower K values
    for factor, multiplier in {"half": 0.5, "double": 2}.items():
        km = KMeans(k=int(k * multiplier))
        km.fit(data)
        preds = km.predict(data)

        results[f"{name}_{factor}"] = {
            "Objective": kmeans_objective(data, preds, km.centroids),
            "Purity": purity_score(labels, preds),
            "Gini Index": gini_index(labels, preds)
        }

print(results)
```

```
/var/folders/2d/kjz0bk3s5nj4p4v10t35f8f40000gn/T/ipykernel_8663914398856
53.py:29: RuntimeWarning: invalid value encountered in divide
  proportions = cluster / np.sum(cluster)
/var/folders/2d/kjz0bk3s5nj4p4v10t35f8f40000gn/T/ipykernel_8663914398856
53.py:29: RuntimeWarning: invalid value encountered in divide
  proportions = cluster / np.sum(cluster)
/var/folders/2d/kjz0bk3s5nj4p4v10t35f8f40000gn/T/ipykernel_8663914398856
53.py:29: RuntimeWarning: invalid value encountered in divide
  proportions = cluster / np.sum(cluster)
```

{'MNIST': {'Objective': 8077677453.634104, 'Purity': 0.569, 'Gini Index': 0.5596332793331051}, 'MNIST_half': {'Objective': 9482325314.473421, 'Purity': 0.452, 'Gini Index': nan}, 'MNIST_double': {'Objective': 6720295577.77635, 'Purity': 0.7106, 'Gini Index': 0.40856988453190235}, 'FASHION': {'Objective': 6507857495.657722, 'Purity': 0.5398, 'Gini Index': 0.5649251411556635}, 'FASHION_half': {'Objective': 8819454308.01257, 'Purity': 0.4188, 'Gini Index': nan}, 'FASHION_double': {'Objective': 4807882275.618217, 'Purity': 0.65, 'Gini Index': 0.45012659919913156}, '20NG': {'Objective': 270.58083294772496, 'Purity': 0.2878, 'Gini Index': 0.7968240834059621}, '20NG_half': {'Objective': 331.15340506797617, 'Purity': 0.2166, 'Gini Index': nan}, '20NG_double': {'Objective': 219.42475357447225, 'Purity': 0.319, 'Gini Index': 0.770127727306173}}

PROBLEM 3 : Gaussian Mixture on toy data

In [15]:

```
import numpy as np
from scipy.stats import multivariate_normal

def initialize_parameters(data, n_components):
    np.random.seed(42)  # for reproducibility
    n_samples, _ = data.shape
    shuffled_indices = np.random.permutation(n_samples)
    means = data[shuffled_indices[:n_components]]
    covariances = [np.cov(data, rowvar=False)] * n_components
    weights = [1./n_components] * n_components
    return means, covariances, weights

def e_step(data, means, covs, weights):
    n_samples, n_features = data.shape
    n_components = len(weights)
    resp = np.zeros((n_samples, n_components))
    for i in range(n_components):
        resp[:, i] = weights[i] * multivariate_normal(mean=means[i], cov=co
    resp /= resp.sum(axis=1)[:, np.newaxis]
    return resp

def m_step(data, resp):
    n_samples, n_features = data.shape
    n_components = resp.shape[1]
    weights = resp.sum(axis=0) / n_samples
    means = np.dot(resp.T, data) / resp.sum(axis=0)[:, np.newaxis]
    covariances = []
    for i in range(n_components):
        diff = (data - means[i]).T
        cov = np.dot(resp[:, i] * diff, diff.T) / resp[:, i].sum()
        covariances.append(cov)
    return means, covariances, weights

def compute_log_likelihood(data, means, covs, weights):
    n_samples = data.shape[0]
    n_components = len(weights)
    log_likelihood = np.zeros((n_samples, n_components))
    for i in range(n_components):
        log_likelihood[:, i] = np.log(weights[i]) + multivariate_normal(mea
    return log_likelihood.sum()

def em_gmm(data, n_components, max_iter=100, tol=1e-4):
    means, covs, weights = initialize_parameters(data, n_components)
    log_likelihood = -np.inf
    for i in range(max_iter):
        resp = e_step(data, means, covs, weights)
        means, covs, weights = m_step(data, resp)
        new_log_likelihood = compute_log_likelihood(data, means, covs, weig
        if np.abs(new_log_likelihood - log_likelihood) < tol:
            break
        log_likelihood = new_log_likelihood
    return means, covs, weights

# Load data
data_2gaussian = np.loadtxt('2gaussian.txt')
data_3gaussian = np.loadtxt('3gaussian.txt')
```

```python
# Run EM for 2-Gaussian data
means_2g, covs_2g, weights_2g = em_gmm(data_2gaussian, 2)
print("Results for 2-gaussian.txt")
print("Means:", means_2g)
print("Covariances:", covs_2g)
print("Weights:", weights_2g)

# Run EM for 3-Gaussian data
means_3g, covs_3g, weights_3g = em_gmm(data_3gaussian, 3)
print("\nResults for 3-gaussian.txt")
print("Means:", means_3g)
print("Covariances:", covs_3g)
print("Weights:", weights_3g)
```

```
Results for 2-gaussian.txt
Means: [[2.99413182 3.0520966 ]
 [7.01314831 3.98313418]]
Covariances: [array([[1.01023425, 0.02719139],
       [0.02719139, 2.93782297]]), array([[0.97475893, 0.49747031],
       [0.49747031, 1.0011426 ]])]
Weights: [0.33479577 0.66520423]

Results for 3-gaussian.txt
Means: [[5.01177664 7.00151504]
 [3.03980012 3.04879137]
 [7.02158525 4.01547353]]
Covariances: [array([[0.97965466, 0.18512102],
       [0.18512102, 0.97448689]]), array([[1.02858546, 0.02702432],
       [0.02702432, 3.38530379]]), array([[0.99037288, 0.50094401],
       [0.50094401, 0.99564696]])]
Weights: [0.49594551 0.20562097 0.29843352]
```

PROBLEM 4 EM for coin flips

In [17]:
```
import numpy as np
from scipy.stats import binom

# Load the data from the file
with open('coin_flips_outcome.txt', 'r') as file:
    data = [list(map(int, line.strip().split())) for line in file.readlines

data = np.array(data)

N, D = data.shape

# Initialize parameters
p = np.random.rand(3)
pi = np.random.rand(3)
pi /= pi.sum()

def e_step(data, p, pi):
    weights = np.zeros((N, 3))

    for i in range(N):
        for j in range(3):
            weights[i, j] = pi[j] * binom.pmf(np.sum(data[i]), D, p[j])

    weights /= weights.sum(axis=1, keepdims=True)

    return weights

def m_step(data, weights):
    p = np.sum(weights * np.sum(data, axis=1, keepdims=True), axis=0) / (D
    pi = np.mean(weights, axis=0)
    return p, pi

# EM algorithm
max_iterations = 1000
tolerance = 1e-6
for iteration in range(max_iterations):
    # E-step
    weights = e_step(data, p, pi)

    # M-step
    p_new, pi_new = m_step(data, weights)

    # Check for convergence
    if np.max(abs(p_new - p)) < tolerance and np.max(abs(pi_new - pi)) < to
        break

    p, pi = p_new, pi_new

print("p_A, p_B, p_C:", p)
print("pi_A, pi_B, pi_C:", pi)
```

```
p_A, p_B, p_C: [0.9317285  0.23691765 0.610037  ]
pi_A, pi_B, pi_C: [0.1785578  0.30681208 0.51463012]
```