Assignment-2 Codes

Team : 18

Dataset 1A

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix


train_data = pd.read_csv('data/Dataset 1A/train.csv')
train_data = train_data.to_numpy()

dev_data = pd.read_csv('data/Dataset 1A/dev.csv')

dev_data = dev_data.to_numpy()


def get_class_wise_data(data):
    obj = {}
    d = len(data[0]) - 1
    for point in data:
        curr_class = point[d]
        if obj.get(curr_class) is None:
            obj[curr_class] = [point[:d]]
        else:
            obj.get(curr_class).append(point[:d])
    for class_ in obj:
        obj[class_] = np.array(obj.get(class_))
    return obj


def get_knns(k, x, data):
    distances = []
    x_coords = x[:len(x)-1]
    for point in data:
        if (point == x).all():  # don't check with same point
            continue
```

```
        point_coords = point[:len(point)-1]  # last element is the class
        dist = np.linalg.norm(x_coords - point_coords)  # l2 norm
        distances.append((dist, point_coords, point[len(point)-1]))
    distances.sort(key=lambda a: a[0])
    top_k_points = distances[:k]
    class_counts = {}  # key will be class, and value will be number of points belonging to the
class
    for point in top_k_points:
        if class_counts.get(point[2]) is None:
            class_counts[point[2]] = 1
        else:
            class_counts[point[2]] = class_counts.get(point[2]) + 1
    pred = 0
    max_count = 0
    for class_ in class_counts:
        if class_counts.get(class_) > max_count:
            pred = class_
            max_count = class_counts.get(class_)
    return pred


def get_knn_accuracy(train_, val_, k_vals=[1]):
    accuracies = {}
    for k in k_vals:
        print(k)
        correct = 0
        total = 0
        y_true = []
        y_pred = []
        for point in val_:
            pred = get_knns(k, point, train_)
            y_pred.append(pred)
            y_true.append(point[len(point)-1])
            if pred == point[len(point)-1]:
                correct += 1
            total += 1
        acc = float(correct / total)
        accuracies[k] = acc
        print(confusion_matrix(y_true, y_pred))
    return accuracies


def naive_bayes_classifier(data, case=1):
    assert 1 <= case <= 3
```

```
    class_wise_data = {}
    d = len(data[0]) - 1
    for point in data:
        class_ = point[len(point)-1]
        if class_wise_data.get(class_) is None:
            class_wise_data[class_] = [point[:d]]
        else:
            class_wise_data.get(class_).append(point[:d])
    num_classes = len(class_wise_data)
    class_wise_mean_var = {}
    for class_ in class_wise_data:
        class_points = np.array(class_wise_data.get(class_), dtype=object)
        curr_mean = class_points.mean(axis=0)
        curr_variance = class_points.var(axis=0)
        class_wise_mean_var[class_] = (curr_mean, curr_variance)
    res = {}
    priors = {}
    if case == 1:  # cov matrix = sigma^2 * I
        sigma = 0
        for class_ in class_wise_mean_var:
            sigma += class_wise_mean_var.get(class_)[1].sum()
        sigma = sigma / (num_classes * d)
        cov_matrix = sigma * np.identity(d)
        for class_ in class_wise_mean_var:
            res[class_] = (class_wise_mean_var.get(class_)[0], cov_matrix)
            priors[class_] = len(class_wise_data.get(class_)) / len(data)
    elif case == 2:  # cov_matrix = C
        cov_matrix = np.zeros((d, d))
        for class_ in class_wise_mean_var:
            class_points = np.array(class_wise_data.get(class_), dtype=object)
            class_mean = class_wise_mean_var.get(class_)[0]
            mean_subtracted = class_points - class_mean
            class_cov_matrix = (np.matmul(mean_subtracted.T, mean_subtracted)) /
len(class_points)
            cov_matrix = np.add(cov_matrix, class_cov_matrix)
        cov_matrix = cov_matrix / num_classes
        for class_ in class_wise_mean_var:
            res[class_] = (class_wise_mean_var.get(class_)[0], cov_matrix)
            priors[class_] = len(class_wise_data.get(class_)) / len(data)
    elif case == 3:
        for class_ in class_wise_mean_var:
            class_points = np.array(class_wise_data.get(class_), dtype=object)
            class_mean = class_wise_mean_var.get(class_)[0]
            mean_subtracted = class_points - class_mean
```

```
            class_cov_matrix = (np.matmul(mean_subtracted.T, mean_subtracted)) /
len(class_points)
            res[class_] = (class_wise_mean_var.get(class_)[0], class_cov_matrix)
            priors[class_] = len(class_wise_data.get(class_)) / len(data)
    return res, priors


def predict_naive_bayes(mean_var, priors, x):
    pred = 0
    max_prob = 0.0
    d = len(x)
    for class_ in mean_var:
        class_mean = mean_var.get(class_)[0]
        class_cov = mean_var.get(class_)[1]
        class_cov = class_cov.astype(np.float64)
        x_mean_sub = x - class_mean
        exp_pow = np.dot(x_mean_sub, np.dot(np.linalg.inv(class_cov), x_mean_sub.T))
        exp_pow = (-1/2) * exp_pow
        cov_det = np.linalg.det(class_cov)
        p = np.exp(exp_pow)
        p /= (2 * np.pi) ** (d / 2)
        p /= np.sqrt(cov_det)
        if p > max_prob:
            max_prob = p
            pred = class_
    return pred


def get_naive_bayes_accuracy(mean_var, priors, data):
    correct = 0
    y_true = []
    y_pred = []
    for point in data:
        pred = predict_naive_bayes(mean_var, priors, point)
        y_pred.append(pred)
        y_true.append(point[len(point)-1])
        if pred == point[len(point)-1]:
            correct += 1
    print(confusion_matrix(y_true, y_pred))
    acc = correct / len(data)
    return acc


# accs = get_knn_accuracy(train_data, train_data, k_vals=[1, 7, 15])
```

```
# for key in accs:
#     print('K:', key, 'Accuracy:', accs.get(key))


# cases = [1, 2, 3]
# for case in cases:
#     mean_vars, priors = naive_bayes_classifier(train_data, case=case)
#     print('Case:', case, 'Accuracy:', get_naive_bayes_accuracy(mean_vars, priors, dev_data))

mean_vars, priors = naive_bayes_classifier(train_data, case=2)
class_wise_data = get_class_wise_data(train_data)
fig, ax = plt.subplots()
colors = ['r', 'g', 'b', 'y']
legend_arr = []
i = 0
min_x = 999
max_x = -999
min_y = 999
max_y = -999
for class_ in class_wise_data:
    class_data = class_wise_data.get(class_)
    for point in class_data:
        if point[0] < min_x:
            min_x = point[0]
        if point[0] > max_x:
            max_x = point[0]
        if point[1] < min_y:
            min_y = point[1]
        if point[1] > max_y:
            max_y = point[1]
    # ax.scatter(x, y, color=colors[i])
    i += 1
# plt.legend(legend_arr)
# plt.show()

# mean_vars, priors = naive_bayes_classifier(train_data, case=2)
x_list = np.linspace(min_x, max_x, 100)
y_list = np.linspace(min_y, max_y, 100)


k = 0
for class_ in mean_vars:
    z = np.zeros((len(x_list), len(x_list)))
    X = []
```

```
    Y = []
    for i in range(len(x_list)):
        temp_x = []
        temp_y = []
        for j in range(len(y_list)):
            temp_x.append(x_list[i])
            temp_y.append(y_list[j])
            point = np.array([x_list[i], y_list[j]])
            d = len(point)
            class_mean = mean_vars.get(class_)[0]
            class_cov = mean_vars.get(class_)[1]
            class_cov = class_cov.astype(np.float64)
            x_mean_sub = point - class_mean
            exp_pow = np.dot(x_mean_sub, np.dot(np.linalg.inv(class_cov), x_mean_sub.T))
            exp_pow = (-1 / 2) * exp_pow
            cov_det = np.linalg.det(class_cov)
            p = np.exp(exp_pow)
            p /= (2 * np.pi) ** (d / 2)
            p /= np.sqrt(cov_det)
            z[i][j] = p
        X.append(temp_x)
        Y.append(temp_y)

    ax.contour(X, Y, z, 10, colors=colors[k])
    k += 1

for class_ in class_wise_data:
    x = []
    y = []
    legend_arr.append(class_)
    class_data = class_wise_data.get(class_)
    for point in class_data:
        x.append(point[0])
        y.append(point[1])
    ax.scatter(x, y, color='k')
    i += 1

plt.show()
```

Dataset 1B

```
import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from scipy.stats import multivariate_normal


train_data = pd.read_csv('data/Dataset 1B/train.csv')
train_data = train_data.to_numpy()

val_data = pd.read_csv('data/Dataset 1B/dev.csv')
val_data = val_data.to_numpy()


def get_class_wise_data(data):
    obj = {}
    d = len(data[0]) - 1
    for point in data:
        curr_class = point[d]
        if obj.get(curr_class) is None:
            obj[curr_class] = [point[:d]]
        else:
            obj.get(curr_class).append(point[:d])
    for class_ in obj:
        obj[class_] = np.array(obj.get(class_))
    return obj


def get_kmeans(data, k, num_iters=10, diagonal=False):
    d = data.shape[1]
    init_means = np.random.randint(0, data.shape[0], size=k)
    kmeans = np.zeros((k, d))
    for i in range(k):
        kmeans[i] = data[init_means[i]]
    point_means = np.zeros(data.shape)
    tol = 0.001
    for i in range(num_iters):
        for j in range(data.shape[0]):
            point = data[j]
            min_dist = 999
            curr_mean = kmeans[0]
```

```
        for k in range(kmeans.shape[0]):
            curr_dist = np.linalg.norm(point - kmeans[k])
            if curr_dist < min_dist:
                curr_mean = kmeans[k]
                min_dist = curr_dist
        point_means[j] = curr_mean
    err = 0
    for j in range(kmeans.shape[0]):
        curr_sum = np.zeros(data.shape[1])
        curr_num = 0
        for k in range(data.shape[0]):
            if (point_means[k] == kmeans[j]).all():
                curr_sum += data[k]
                curr_num += 1
        new_mean = curr_sum / curr_num
        err += np.linalg.norm(kmeans[j] - new_mean)
        kmeans[j] = new_mean
    if err < tol:
        break

for j in range(data.shape[0]):
    point = data[j]
    min_dist = 999
    curr_mean = kmeans[0]
    for k in range(kmeans.shape[0]):
        curr_dist = np.linalg.norm(point - kmeans[k])
        if curr_dist < min_dist:
            curr_mean = kmeans[k]
            min_dist = curr_dist
    point_means[j] = curr_mean

covmatrices = []
wqs = []
for j in range(kmeans.shape[0]):
    data_points = []
    curr_count = 0
    for k in range(data.shape[0]):
        if (point_means[k] == kmeans[j]).all():
            data_points.append(data[k])
            curr_count += 1
```

```
        data_points = np.array(data_points)
        data_points = data_points - kmeans[j]
        cov_matrix = np.matmul(data_points.T, data_points)
        if diagonal:
            for x in range(cov_matrix.shape[0]):
                for y in range(cov_matrix.shape[1]):
                    if x != y:
                        cov_matrix[x][y] = 0.0
        covmatrices.append(cov_matrix)
        wqs.append(curr_count / data.shape[0])
    covmatrices = np.array(covmatrices)
    wqs = np.array(wqs)
    return kmeans, covmatrices, wqs


def calculate_responsibilty_terms(data, kmeans, covmatrices, wqs):
    Q = kmeans.shape[0]
    gammas = np.zeros((data.shape[0], Q))
    d = len(data[0]) - 1
    for i in range(data.shape[0]):
        den = 0
        for j in range(Q):
            mean_subtracted = data[i] - kmeans[j]
            cov_inv = np.linalg.inv(covmatrices[j])
            cov_det = np.linalg.det(covmatrices[j])
            prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
            prod = -0.5 * prod
            p = np.exp(prod)
            p /= (2 * np.pi) ** (d / 2)
            p /= np.sqrt(cov_det)
            p = wqs[j] * p
            den += p
            gammas[i][j] = p
        gammas[i] /= den
    gammas = np.array(gammas)
    return gammas


def maximization_step(data, gammas, diagonal=False):
    d = data.shape[1]
```

```
    q = gammas.shape[1]
    nq = np.sum(gammas, axis=0)
    wqs = nq / data.shape[0]
    mu_q = np.zeros((q, d))
    for i in range(q):
        curr = np.zeros(d)
        for j in range(data.shape[0]):
            curr += gammas[j][i] * data[j]
        mu_q[i] = curr / nq[i]
    c_q = []
    for i in range(q):
        curr = np.zeros((d, d))
        for j in range(data.shape[0]):
            mean_subtracted = data[j] - mu_q[i]
            mean_subtracted = np.array([mean_subtracted])
            curr += gammas[j][i] * np.multiply(mean_subtracted.T, mean_subtracted)
        curr /= nq[i]
        if diagonal:
            for x in range(curr.shape[0]):
                for y in range(curr.shape[1]):
                    if x != y:
                        curr[x][y] = 0.0
        c_q.append(curr)
    c_q = np.array(c_q)
    return mu_q, c_q, wqs


def calculate_log_likelihood(data, mus, cqs, wqs):
    likelihood = 0
    q = wqs.shape[0]
    d = data.shape[1]
    for i in range(data.shape[0]):
        curr = 0
        for j in range(q):
            mean_subtracted = data[i] - mus[j]
            cov_inv = np.linalg.inv(cqs[j])
            cov_det = np.linalg.det(cqs[j])
            prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
            prod = -0.5 * prod
            p = np.exp(prod)
```

```
            p /= (2 * np.pi) ** (d / 2)
            p /= np.sqrt(cov_det)
            curr += wqs[j] * p
        likelihood += np.log(curr)
    return likelihood


def gmm(data, q=4, diagonal=False):
    print('Making model...')
    class_wise_data = get_class_wise_data(data)
    tol = 0.001
    res = {}
    for class_ in class_wise_data:
        k_means, cov_matrices, w_qs = get_kmeans(class_wise_data.get(class_), q,
diagonal=diagonal)
        curr_likelihood = calculate_log_likelihood(class_wise_data.get(class_), k_means,
cov_matrices, w_qs)
        err = 999
        while err > tol:
            gammas_ = calculate_responsibilty_terms(class_wise_data.get(class_),
k_means, cov_matrices, w_qs)
            k_means, cov_matrices, w_qs =
maximization_step(class_wise_data.get(class_), gammas_, diagonal=diagonal)
            new_likelihood = calculate_log_likelihood(class_wise_data.get(class_),
k_means, cov_matrices, w_qs)
            err = abs(new_likelihood - curr_likelihood)
            curr_likelihood = new_likelihood
        res[class_] = {
            'mu_q': k_means,
            'c_q': cov_matrices,
            'w_q': w_qs
        }
    return res


def calculate_gaussian_probabilty(x, mu, cov_matrix):
    d = len(x)
    mean_subtracted = x - mu
    cov_inv = np.linalg.inv(cov_matrix)
    cov_det = np.linalg.det(cov_matrix)
```

```
    prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
    prod = -0.5 * prod
    p = np.exp(prod)
    p /= (2 * np.pi) ** (d / 2)
    p /= np.sqrt(cov_det)
    return p


def get_probabilty(x, mus, cqs, wqs):
    prob = 0
    q = len(mus)
    for i in range(q):
        p = calculate_gaussian_probabilty(x, mus[i], cqs[i])
        prob += wqs[i] * p
    return prob


def predict(x, model):
    pred = 0
    max_prob = 0
    for class_ in model:
        class_params = model.get(class_)
        muq = class_params.get('mu_q')
        cq = class_params.get('c_q')
        wqs = class_params.get('w_q')
        curr_prob = get_probabilty(x, muq, cq, wqs)
        if curr_prob > max_prob:
            pred = class_
            max_prob = curr_prob
    return pred


def get_accuracy(data, model):
    print('Getting accuracy...')
    correct = 0
    total = data.shape[0]
    d = data.shape[1] - 1
    y_true = []
    y_pred = []
    for point in data:
```

```
            correct_class = point[d]
            pred_class = predict(point[:d], model)
            y_pred.append(pred_class)
            y_true.append(correct_class)
            if pred_class == correct_class:
                correct += 1
        acc = correct / total
        return acc, y_true, y_pred


def knn_classifier(x, data, k):
    pred = 0
    min_radius = 999
    for class_ in data:
        class_data = data.get(class_)
        distances = []
        for point in class_data:
            if (point == x).all():
                continue
            dist = np.linalg.norm(x - point)
            distances.append(dist)
        distances.sort()
        r_i = distances[k-1]
        if r_i < min_radius:
            pred = class_
            min_radius = r_i
    return pred


def get_knn_accuracy(train_, val_, k):
    class_wise_acc = {}
    y_pred = []
    y_true = []
    for class_ in val_:
        class_points = val_.get(class_)
        correct = 0
        total = 0
        for point in class_points:
            pred = knn_classifier(point, train_, k)
            y_pred.append(pred)
```

```
            y_true.append(class_)
            total += 1
            if pred == class_:
                correct += 1
        acc = correct / total
        class_wise_acc[class_] = {
            'correct': correct,
            'total': total,
            'accuracy': acc
        }
    return class_wise_acc, y_true, y_pred


# GMM model
# Qs = [4, 5, 6, 7]
# for x in Qs:
#     model = gmm(train_data, q=x, diagonal=False)  # diagonal = True for diagonal
covariance matrix
#     train_acc, train_true, train_pred = get_accuracy(train_data, model)
#     val_acc, val_true, val_pred = get_accuracy(val_data, model)
#     print('Q:', x, 'Train Accuracy:', train_acc)
#     print('Q:', x, 'Val Accuracy:', val_acc)
#     print('Train confusion matrix:')
#     print(confusion_matrix(train_true, train_pred))
#     print('Validation Confusion Matrix:')
#     print(confusion_matrix(val_true, val_pred))

# KNN model
# train_class_wise = get_class_wise_data(train_data)
# val_class_wise = get_class_wise_data(val_data)
# Ks = [10, 20]
# for k in Ks:
#     train_res, train_true, train_pred = get_knn_accuracy(train_class_wise,
train_class_wise, k)
#     val_res, val_true, val_pred = get_knn_accuracy(train_class_wise, val_class_wise,
k)
#     print('K:', k)
#     total = 0.0
#     val_total = 0.0
#     for x in train_res:
```

```
#         print('Class:', x, 'Train Accuracy:', train_res.get(x).get('accuracy'))
#         print('Class:', x, 'Val Accuracy:', val_res.get(x).get('accuracy'))
#         total += train_res.get(x).get('accuracy')
#         val_total += val_res.get(x).get('accuracy')
#     total /= len(train_res)
#     val_total /= len(val_res)
#     print('Train Accuracy:', total)
#     print('Val Accuracy:', val_total)
#     print('Train Confusion Matrix:')
#     print(confusion_matrix(train_true, train_pred))
#     print('Val Confusion Matrix:')
#     print(confusion_matrix(val_true, val_pred))


fig, ax = plt.subplots()
colors = ['r', 'g', 'b', 'y']
# legend_arr = []
# i = 0
min_x = 999
max_x = -999
min_y = 999
max_y = -999
class_wise_data = get_class_wise_data(train_data)
for class_ in class_wise_data:
    class_data = class_wise_data.get(class_)
    for point in class_data:
        if point[0] < min_x:
            min_x = point[0]
        if point[0] > max_x:
            max_x = point[0]
        if point[1] < min_y:
            min_y = point[1]
        if point[1] > max_y:
            max_y = point[1]
    # ax.scatter(x, y, color=colors[i])
# # plt.legend(legend_arr)
# # plt.show()
#
# x_list = np.linspace(min_x, max_x, 100)
# y_list = np.linspace(min_y, max_y, 100)
```

```
#
# z = np.zeros((len(x_list), len(x_list)))
# X = []
# Y = []
#
# for i in range(len(x_list)):
#     temp_x = []
#     temp_y = []
#     for j in range(len(y_list)):
#         temp_x.append(x_list[i])
#         temp_y.append(y_list[j])
#         point = np.array([x_list[i], y_list[j]])
#         z[i][j] = knn_classifier(point, class_wise_data, 10)
#     X.append(temp_x)
#     Y.append(temp_y)
#
# ax.contourf(X, Y, z)
#
# for class_ in class_wise_data:
#     x = []
#     y = []
#     legend_arr.append(class_)
#     class_data = class_wise_data.get(class_)
#     for point in class_data:
#         x.append(point[0])
#         y.append(point[1])
#     ax.scatter(x, y, color='k')
#
# plt.show()
model = gmm(train_data, q=7, diagonal=True)  # diagonal = True for diagonal
covariance matrix

x_list = np.linspace(min_x, max_x, 100)
y_list = np.linspace(min_y, max_y, 100)

k = 0
for class_ in model:
    z = np.zeros((len(x_list), len(x_list)))
    X = []
    Y = []
```

```
    for i in range(len(x_list)):
        temp_x = []
        temp_y = []
        for j in range(len(y_list)):
            temp_x.append(x_list[i])
            temp_y.append(y_list[j])
            point = np.array([x_list[i], y_list[j]])
            z[i][j] = get_probabilty(point, model.get(class_).get('mu_q'),
model.get(class_).get('c_q'), model.get(class_).get('w_q'))
        X.append(temp_x)
        Y.append(temp_y)

    ax.contour(X, Y, z, 38, colors=colors[k])
    k += 1

for class_ in class_wise_data:
    x = []
    y = []
    class_data = class_wise_data.get(class_)
    for point in class_data:
        x.append(point[0])
        y.append(point[1])
    ax.scatter(x, y, color='k')

plt.show()
```

Dataset 2A

Bayes classifier with a GMM for each class, using full/diagonal covariance matrices

```
import numpy as np
import pandas as pd
import pickle
from sklearn.metrics import confusion_matrix

def get_class_wise_data(data):
    obj = {}
    d = len(data[0]) - 1
    for point in data:
        curr_class = point[d]
```

```
        if obj.get(curr_class) is None:
            obj[curr_class] = [point[:d]]
        else:
            obj.get(curr_class).append(point[:d])
    for class_ in obj:
        obj[class_] = np.array(obj.get(class_))
    return obj


def get_kmeans(data, k, num_iters=10, diagonal=False):
    d = data.shape[1]
    init_means = np.random.randint(0, data.shape[0], size=k)
    kmeans = np.zeros((k, d))
    for i in range(k):
        kmeans[i] = data[init_means[i]]
    point_means = np.zeros(data.shape)
    tol = 0.001
    for i in range(num_iters):
        for j in range(data.shape[0]):
            point = data[j]
            min_dist = 999
            curr_mean = kmeans[0]
            for k in range(kmeans.shape[0]):
                curr_dist = np.linalg.norm(point - kmeans[k])
                if curr_dist < min_dist:
                    curr_mean = kmeans[k]
                    min_dist = curr_dist
            point_means[j] = curr_mean
        err = 0
        for j in range(kmeans.shape[0]):
            curr_sum = np.zeros(data.shape[1])
            curr_num = 0
            for k in range(data.shape[0]):
                if (point_means[k] == kmeans[j]).all():
                    curr_sum += data[k]
                    curr_num += 1
            new_mean = curr_sum / curr_num
            err += np.linalg.norm(kmeans[j] - new_mean)
            kmeans[j] = new_mean
        if err < tol:
```

```
            break

    for j in range(data.shape[0]):
        point = data[j]
        min_dist = 999
        curr_mean = kmeans[0]
        for k in range(kmeans.shape[0]):
            curr_dist = np.linalg.norm(point - kmeans[k])
            if curr_dist < min_dist:
                curr_mean = kmeans[k]
                min_dist = curr_dist
        point_means[j] = curr_mean

    covmatrices = []
    wqs = []
    for j in range(kmeans.shape[0]):
        data_points = []
        curr_count = 0
        for k in range(data.shape[0]):
            if (point_means[k] == kmeans[j]).all():
                data_points.append(data[k])
                curr_count += 1
        data_points = np.array(data_points)
        data_points = data_points - kmeans[j]
        cov_matrix = np.matmul(data_points.T, data_points)
        if diagonal:
            for x in range(cov_matrix.shape[0]):
                for y in range(cov_matrix.shape[1]):
                    if x != y:
                        cov_matrix[x][y] = 0.0
        covmatrices.append(cov_matrix)
        wqs.append(curr_count / data.shape[0])
    covmatrices = np.array(covmatrices)
    wqs = np.array(wqs)
    return kmeans, covmatrices, wqs


def calculate_responsibilty_terms(data, kmeans, covmatrices, wqs, mult):
    Q = kmeans.shape[0]
    gammas = np.zeros((data.shape[0], Q))
```

```
    d = len(data[0]) - 1
    for i in range(data.shape[0]):
        den = 0
        for j in range(Q):
            mean_subtracted = data[i] - kmeans[j]
            cov_det = np.linalg.det(covmatrices[j])
            if cov_det<0.0001:
                dim = len(covmatrices[j])
                covmatrices[j] += mult*np.identity(dim)
            cov_inv = np.linalg.pinv(covmatrices[j], rcond=1e-06)
            prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
            prod = -0.5 * prod
            p = np.exp(prod)
            p /= (2 * np.pi) ** (d / 2)
            p /= np.sqrt(cov_det)
            p = wqs[j] * p
            den += p
            gammas[i][j] = p
        if den == 0:
            den = 1e-300
        gammas[i] /= den
    gammas = np.array(gammas)
    return gammas


def maximization_step(data, gammas, diagonal=False):
    d = data.shape[1]
    q = gammas.shape[1]
    nq = np.sum(gammas, axis=0)
    wqs = nq / data.shape[0]
    mu_q = np.zeros((q, d))
    for i in range(q):
        curr = np.zeros(d)
        for j in range(data.shape[0]):
            curr += gammas[j][i] * data[j]
        mu_q[i] = curr / nq[i]
    c_q = []
    for i in range(q):
        curr = np.zeros((d, d))
        for j in range(data.shape[0]):
```

```
            mean_subtracted = data[j] - mu_q[i]
            mean_subtracted = np.array([mean_subtracted])
            curr += gammas[j][i] * np.multiply(mean_subtracted.T, mean_subtracted)
        curr /= nq[i]
        if diagonal:
            for x in range(curr.shape[0]):
                for y in range(curr.shape[1]):
                    if x != y:
                        curr[x][y] = 0.0
        c_q.append(curr)
    c_q = np.array(c_q)
    return mu_q, c_q, wqs


def calculate_log_likelihood(data, mus, cqs, wqs, mult):
    likelihood = 0
    q = wqs.shape[0]
    d = data.shape[1]
    for i in range(data.shape[0]):
        curr = 0
        for j in range(q):
            mean_subtracted = data[i] - mus[j]
            cov_det = np.linalg.det(cqs[j])
            if cov_det<0.0001:
                dim = len(cqs[j])
                cqs[j] += mult*np.identity(dim)
            cov_inv = np.linalg.pinv(cqs[j], rcond=1e-06)
            prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
            prod = -0.5 * prod
            p = np.exp(prod)
            p /= (2 * np.pi) ** (d / 2)
            p /= np.sqrt(cov_det)
            curr += wqs[j] * p
        likelihood += np.log(curr)
    return likelihood


def gmm(class_wise_data, q=4, mult=0.001, diagonal=False):
    print('Making model...')
    tol = 0.001
```

```
    res = {}
    for class_ in class_wise_data:
        k_means, cov_matrices, w_qs = get_kmeans(class_wise_data.get(class_), q,
diagonal=diagonal)
        curr_likelihood = calculate_log_likelihood(class_wise_data.get(class_), k_means,
cov_matrices, w_qs, mult)
        err = 999
        while err > tol:
            gammas_ = calculate_responsibilty_terms(class_wise_data.get(class_),
k_means, cov_matrices, w_qs, mult)
            k_means, cov_matrices, w_qs =
maximization_step(class_wise_data.get(class_), gammas_, diagonal=diagonal)
            new_likelihood = calculate_log_likelihood(class_wise_data.get(class_),
k_means, cov_matrices, w_qs, mult)
            err = abs(new_likelihood - curr_likelihood)
            curr_likelihood = new_likelihood
        res[class_] = {
            'mu_q': k_means,
            'c_q': cov_matrices,
            'w_q': w_qs
        }
    return res


def calculate_gaussian_probabilty(x, mu, cov_matrix, mult):
    d = len(x)
    dim = len(cov_matrix)
    mean_subtracted = x - mu
    cov_det = np.linalg.det(cov_matrix)
    if cov_det<0.0001:
        cov_matrix += mult*np.identity(dim)
    cov_inv = np.linalg.pinv(cov_matrix, rcond=1e-06)
    prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
    prod = -0.5 * prod
    p = np.exp(prod)
    p /= (2 * np.pi) ** (d / 2)
    p /= np.sqrt(cov_det)
    return p
```

```
def get_probabilty(x, mus, cqs, wqs, mult):
    prob = 0
    q = len(mus)
    for i in range(q):
        p = calculate_gaussian_probabilty(x, mus[i], cqs[i], mult)
        prob += wqs[i] * p
    return prob


def predict(x, model,mult):
    pred = 0
    max_prob = 0
    for class_ in model:
        class_params = model.get(class_)
        muq = class_params.get('mu_q')
        cq = class_params.get('c_q')
        wqs = class_params.get('w_q')
        curr_prob = get_probabilty(x, muq, cq, wqs, mult)
        if curr_prob > max_prob:
            pred = class_
            max_prob = curr_prob
    return pred


def get_accuracy(data, label, model,mult):
    print('Getting accuracy...')
    correct = 0
    total = data.shape[0]
    pred = []
    for i, point in enumerate(data):
        correct_class = label[i]
        pred_class = predict(point, model,mult)
        pred.append(pred_class)
        if pred_class == correct_class:
            correct += 1
    acc = correct / total
    return pred, acc
```

```
# get data for each class
classes = ['coast', 'forest', 'highway', 'mountain', 'tallbuilding']
class_wise_data = {}
data = None
val_data = None
label = []
val_label = []
for i, class_ in enumerate(classes):
    class_train_data = pd.read_csv(class_ + '/train.csv')
    df_class_data = class_train_data.drop(['image_names'], axis=1)
    class_data = df_class_data.to_numpy()

    m,n = class_data.shape
    label += [class_]*m
    if data is None:
        data = class_data
    else:
        data = np.vstack((data, class_data))

    class_wise_data[class_] = class_data

    class_val_data = pd.read_csv(class_ + '/dev.csv')
    dfval_class_data = class_val_data.drop(['image_names'], axis=1)
    class_val_data = dfval_class_data.to_numpy()

    m,n = class_val_data.shape
    val_label += [class_]*m
    if val_data is None:
        val_data = class_val_data
    else:
        val_data = np.vstack((val_data, class_val_data))


Q = [2, 6,12,20,50]
mult = [0.001, 0.01,0.1]


for q in Q:
```

```
    for m in mult:
        # GMM model
        model = gmm(class_wise_data, q, m, diagonal=False)  # diagonal = True for
diagonal covariance matrix
        # save the model to disk
        filename = '2A_q'+str(q)+'_M'+str(m)+'.sav'
        pickle.dump(model, open('Models/' + filename, 'wb'))
        train_pred, train_acc = get_accuracy(data, label, model, m)
        val_pred, val_acc = get_accuracy(val_data, val_label, model, m)
        print('\n\n2A_q'+str(q)+'_M'+str(m))
        print("Q:", q)
        print('Multiplier:', m)
        print('Train Accuracy:', train_acc)
        print('Val Accuracy:', val_acc)
        train_CM = confusion_matrix(label, train_pred, labels=classes)
        print('train confusion matrix:\n', train_CM)
        val_CM = confusion_matrix(val_label, val_pred, labels=classes)
        print('validation confusion matrix:\n', val_CM)
```

Dataset 2B

Bayes classifier with a GMM for each class, using full/diagonal covariance matrices

```
import numpy as np
import os
import pickle
from sklearn.metrics import confusion_matrix

def get_class_wise_data(data):
  obj = {}
  d = len(data[0]) - 1
  for point in data:
     curr_class = point[d]
     if obj.get(curr_class) is None:
        obj[curr_class] = [point[:d]]
     else:
        obj.get(curr_class).append(point[:d])
  for class_ in obj:
     obj[class_] = np.array(obj.get(class_))
  return obj
```

```
def get_kmeans(data, k, num_iters=10, diagonal=False):
    d = data.shape[1]
    init_means = np.random.randint(0, data.shape[0], size=k)
    kmeans = np.zeros((k, d))
    for i in range(k):
        kmeans[i] = data[init_means[i]]
    point_means = np.zeros(data.shape)
    tol = 0.001
    for i in range(num_iters):
        for j in range(data.shape[0]):
            point = data[j]
            min_dist = 999
            curr_mean = kmeans[0]
            for k in range(kmeans.shape[0]):
                curr_dist = np.linalg.norm(point - kmeans[k])
                if curr_dist < min_dist:
                    curr_mean = kmeans[k]
                    min_dist = curr_dist
            point_means[j] = curr_mean
        err = 0
        for j in range(kmeans.shape[0]):
            curr_sum = np.zeros(data.shape[1])
            curr_num = 0
            for k in range(data.shape[0]):
                if (point_means[k] == kmeans[j]).all():
                    curr_sum += data[k]
                    curr_num += 1
            new_mean = curr_sum / curr_num
            err += np.linalg.norm(kmeans[j] - new_mean)
            kmeans[j] = new_mean
        if err < tol:
            break

    for j in range(data.shape[0]):
        point = data[j]
        min_dist = 999
        curr_mean = kmeans[0]
        for k in range(kmeans.shape[0]):
            curr_dist = np.linalg.norm(point - kmeans[k])
            if curr_dist < min_dist:
                curr_mean = kmeans[k]
                min_dist = curr_dist
```

```
            point_means[j] = curr_mean

        covmatrices = []
        wqs = []
        for j in range(kmeans.shape[0]):
            data_points = []
            curr_count = 0
            for k in range(data.shape[0]):
                if (point_means[k] == kmeans[j]).all():
                    data_points.append(data[k])
                    curr_count += 1
            data_points = np.array(data_points)
            data_points = data_points - kmeans[j]
            cov_matrix = np.matmul(data_points.T, data_points)
            if diagonal:
                for x in range(cov_matrix.shape[0]):
                    for y in range(cov_matrix.shape[1]):
                        if x != y:
                            cov_matrix[x][y] = 0.0
            covmatrices.append(cov_matrix)
            wqs.append(curr_count / data.shape[0])
        covmatrices = np.array(covmatrices)
        wqs = np.array(wqs)
        return kmeans, covmatrices, wqs


    def calculate_responsibilty_terms(data, kmeans, covmatrices, wqs, mult):
        Q = kmeans.shape[0]
        gammas = np.zeros((data.shape[0], Q))
        d = len(data[0]) - 1
        for i in range(data.shape[0]):
            den = 0
            for j in range(Q):
                mean_subtracted = data[i] - kmeans[j]
                cov_det = np.linalg.det(covmatrices[j])
                if cov_det<0.0001:
                    dim = len(covmatrices[j])
                    covmatrices[j] += mult*np.identity(dim)
                cov_inv = np.linalg.pinv(covmatrices[j], rcond=1e-06)
                prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
                prod = -0.5 * prod
                p = np.exp(prod)
                p /= (2 * np.pi) ** (d / 2)
                p /= np.sqrt(cov_det)
```

```
            p = wqs[j] * p
            den += p
            gammas[i][j] = p
        if den == 0:
            den = 1e-300
        gammas[i] /= den
    gammas = np.array(gammas)
    return gammas


def maximization_step(data, gammas, diagonal=False):
    d = data.shape[1]
    q = gammas.shape[1]
    nq = np.sum(gammas, axis=0)
    wqs = nq / data.shape[0]
    mu_q = np.zeros((q, d))
    for i in range(q):
        curr = np.zeros(d)
        for j in range(data.shape[0]):
            curr += gammas[j][i] * data[j]
        mu_q[i] = curr / nq[i]
    c_q = []
    for i in range(q):
        curr = np.zeros((d, d))
        for j in range(data.shape[0]):
            mean_subtracted = data[j] - mu_q[i]
            mean_subtracted = np.array([mean_subtracted])
            curr += gammas[j][i] * np.multiply(mean_subtracted.T, mean_subtracted)
        curr /= nq[i]
        if diagonal:
            for x in range(curr.shape[0]):
                for y in range(curr.shape[1]):
                    if x != y:
                        curr[x][y] = 0.0
        c_q.append(curr)
    c_q = np.array(c_q)
    return mu_q, c_q, wqs


def calculate_log_likelihood(data, mus, cqs, wqs, mult):
    likelihood = 0
    q = wqs.shape[0]
    d = data.shape[1]
    for i in range(data.shape[0]):
```

```
        curr = 0
        for j in range(q):
            mean_subtracted = data[i] - mus[j]
            cov_det = np.linalg.det(cqs[j])
            if cov_det<0.0001:
                dim = len(cqs[j])
                cqs[j] += mult*np.identity(dim)
            cov_inv = np.linalg.pinv(cqs[j], rcond=1e-06)
            prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
            prod = -0.5 * prod
            p = np.exp(prod)
            p /= (2 * np.pi) ** (d / 2)
            p /= np.sqrt(cov_det)
            curr += wqs[j] * p
        likelihood += np.log(curr)
    return likelihood


def gmm(class_wise_data, q=4, iter=100, mult=0.001, diagonal=False):
    print('Making model...')
    tol = 0.001
    res = {}
    for class_ in class_wise_data:
        k_means, cov_matrices, w_qs = get_kmeans(class_wise_data.get(class_), q,
diagonal=diagonal)
        curr_likelihood = calculate_log_likelihood(class_wise_data.get(class_), k_means,
cov_matrices, w_qs, mult)
        err = 999
        for i in range(iter):
            gammas_ = calculate_responsibilty_terms(class_wise_data.get(class_), k_means,
cov_matrices, w_qs, mult)
            k_means, cov_matrices, w_qs = maximization_step(class_wise_data.get(class_),
gammas_, diagonal=diagonal)
            new_likelihood = calculate_log_likelihood(class_wise_data.get(class_), k_means,
cov_matrices, w_qs, mult)
            err = abs(new_likelihood - curr_likelihood)
            curr_likelihood = new_likelihood
        res[class_] = {
            'mu_q': k_means,
            'c_q': cov_matrices,
            'w_q': w_qs
        }
    return res
```

```
def calculate_gaussian_probabilty(x, mu, cov_matrix, mult):
    d = len(x)
    dim = len(cov_matrix)
    mean_subtracted = x - mu
    cov_det = np.linalg.det(cov_matrix)
    if cov_det<0.0001:
        cov_matrix += mult*np.identity(dim)
    cov_inv = np.linalg.pinv(cov_matrix, rcond=1e-06)
    prod = np.dot(mean_subtracted, np.dot(cov_inv, mean_subtracted.T))
    prod = -0.5 * prod
    p = np.exp(prod)
    p /= (2 * np.pi) ** (d / 2)
    p /= np.sqrt(cov_det)
    return p


def get_probabilty(x, mus, cqs, wqs, mult):
    prob = 0
    q = len(mus)
    for i in range(q):
        p = calculate_gaussian_probabilty(x, mus[i], cqs[i], mult)
        prob += wqs[i] * p
    return prob


def predict(img, model, mult):
    pred = 0
    max_prob = 0
    for class_ in model:
        class_params = model.get(class_)
        muq = class_params.get('mu_q')
        cq = class_params.get('c_q')
        wqs = class_params.get('w_q')
        curr_prob = 1
        for p in range(len(img)):
            curr_prob *= get_probabilty(img[p], muq, cq, wqs, mult)
        if curr_prob > max_prob:
            pred = class_
            max_prob = curr_prob
    return pred


def get_accuracy(data, label, model, mult):
```

```
    print('Getting accuracy...')
    correct = 0
    total = data.shape[0]
    pred = []
    for i, img in enumerate(data):
        correct_class = label[i]
        pred_class = predict(img, model, mult)
        pred.append(pred_class)
        if pred_class == correct_class:
            correct += 1
    acc = correct / total
    return pred, acc


def knn_classifier(x, data, k):
    pred = 0
    min_radius = 999
    for class_ in data:
        class_data = data.get(class_)
        distances = []
        for point in class_data:
            if (point == x).all():
                continue
            dist = np.linalg.norm(x - point)
            distances.append(dist)
        distances.sort()
        r_i = distances[k-1]
        if r_i < min_radius:
            pred = class_
            min_radius = r_i
    return pred


def get_knn_accuracy(train_, val_, k):
    class_wise_acc = {}
    for class_ in val_:
        class_points = val_.get(class_)
        correct = 0
        total = 0
        for point in class_points:
            pred = knn_classifier(point, train_, k)
            total += 1
            if pred == class_:
                correct += 1
```

```
        acc = correct / total
        class_wise_acc[class_] = {
            'correct': correct,
            'total': total,
            'accuracy': acc
        }
    return class_wise_acc


# get data for each class
classes = ['coast', 'forest', 'highway', 'mountain', 'tallbuilding']
# classes = ['coast']
class_wise_data = {}
data = []
label = []
val_data = []
val_label = []

for i, class_ in enumerate(classes):
    # cwd = os.getcwd()
    path = class_ + '/train/'
    class_img_data = None
    for filename in os.listdir(path):
        f = open(path+filename, 'r')
        img = [line.split() for line in f]
        data.append(np.asarray(img, dtype='float64'))
        label.append(class_)
        if class_img_data is None:
            class_img_data = img
        else:
            class_img_data = np.vstack((class_img_data, img))

    class_img_data = np.asarray(class_img_data, dtype = 'float64')
    # print(class_img_data.shape)
    # m,n = class_img_data.shape
    # label += [class_]*m

    class_wise_data[class_] = class_img_data

        # validation data
    val_path = class_+'/dev/'
    for filename in os.listdir(val_path):
        f_val = open(val_path+filename, 'r')
        val_img = [line.split() for line in f_val]
```

```
        val_data.append(np.asarray(val_img, dtype='float64'))
        val_label.append(class_)

data = np.asarray(data, dtype='float64')
val_data = np.asarray(val_data, dtype='float64')




Q = [6,12,20]
max_iter = [10,20,100]
mult = [0.001, 0.01, 0.1]

for iter in max_iter:
   for q in Q:
      for m in mult:
         # GMM model
         model = gmm(class_wise_data, q, iter, m, diagonal=False)  # diagonal = True for
diagonal covariance matrix
         # save the model to disk
         filename = '2B_q'+str(q)+'_I'+str(iter)+'_M'+str(m)+'.sav'
         pickle.dump(model, open('Models/' + filename, 'wb'))
         train_pred, train_acc = get_accuracy(data, label, model, m)
         val_pred, val_acc = get_accuracy(val_data, val_label, model, m)
         print('\n\n2B_q'+str(q)+'_I'+str(iter)+'_M'+str(m))
         print("Q:", q)
         print("max_iter:", iter)
         print('Multiplier:', m)
         print('Train Accuracy:', train_acc)
         print('Val Accuracy:', val_acc)
         train_CM = confusion_matrix(label, train_pred, labels=classes)
         print('train confusion matrix:\n', train_CM)
         val_CM = confusion_matrix(val_label, val_pred, labels=classes)
         print('validation confusion matrix:\n', val_CM)
```