109550017_HW2_黃品云

Part. 1, Coding (60%):

1. (5%) Compute the mean vectors mi (i=1, 2) of each 2 classes on **training data**

```
mean vector of class 1: [[ 0.99253136 -0.99115481]] mean vector of class 2: [[-0.9888012   1.00522778]]
```

```
# m1, m2 are the mean vector for class 1 and 2

# m1 = (1/N1)*sum(x in C1), N1 = the number of points in C1
# m2 = (1/N2)*sum(x in C2), N2 = the number of points in C2
m1 = np.mean(x_train[y_train == 0], axis = 0)
m2 = np.mean(x_train[y_train == 1], axis = 0)
```

```
print(f"mean vector of class 1: {m1}", f"mean vector of class 2: {m2}")
```

2. (5%) Compute the within-class scatter matrix SW on **training data**

```
Within-class scatter matrix SW: [[ 4337.38546493 -1795.55656547]
 [-1795.55656547  2834.75834886]]
```

```
# SW is the within-class covariance matrix
# SW = sum((x-m1)·(x-m1).T, x in C1) + sum((x-m2)·(x-m2).T, x in C2)

# first, we separate the data into class 1 and 2
x1 = x_train[y_train == 0]
x2 = x_train[y_train == 1]

# SW = sum((x-m1).T·(x-m1), x in C1) + sum((x-m2).T·(x-m2), x in C2)
# I changed the order the transpose to make the shape consistent
SW = np.dot((x1 - m1).T, (x1 - m1)) + np.dot((x2 - m2).T, (x2 - m2))
```

```
print(f"Within-class scatter matrix SW: {SW}")
```

3. (5%) Compute the between-class scatter matrix SB on **training data**

```
Between-class scatter matrix SB: [[ 3.92567873 -3.95549783]
 [-3.95549783  3.98554344]]
```

```
# SB is the between-class covariance matrix
# SB = (m2 - m1)·(m2 - m1).T
# I changed the order the transpose to make the shape consistent

SB = np.dot((m2 - m1).T, (m2 - m1))
```

```
print(f"Between-class scatter matrix SB: {SB}")
```

4. (5%) Compute the Fisher's linear discriminant $_w$ on **training data**

```
Fisher's linear discriminant: [[ 0.37003809]
 [-0.92901658]]
```

```
# The optimal W is the eigenvector of inv(SW)·SB that corresponds to
# the largest eigenvalue

# compute inv(SW)·SB
inv_SW = np.linalg.inv(SW)
A = np.dot(inv_SW, SB)
# get eigenvalue and eigenvector
eigenvalue, eigenvector = np.linalg.eig(A)
# the optimal W is the eigenvector corresponds to the largest eigenvalue
max_eigenvalue_index = np.argmax(eigenvector)
W = eigenvector[:, max_eigenvalue_index]
```

```
print(f" Fisher's linear discriminant: {W}")
```

5. (20%) Project the **testing data** by Fisher's linear discriminant to get the class prediction by K-Nearest-Neighbor rule and report the accuracy score on **testing data** with K values from 1 to 5 (you should get accuracy over 0.9)

```
Accuracy of test-set 0.8488
Accuracy of test-set 0.8488
Accuracy of test-set 0.8792
Accuracy of test-set 0.8824
Accuracy of test-set 0.8912
```

```python
# compute the distance of x1, x2
def euclidean_distance(x1, x2):
    distance = np.sqrt(np.sum((x1 - x2)**2))
    return distance

# for single element
def _predict(t, K):

    # compute the distance
    distances = [euclidean_distance(t, x) for x in train]
    # get the closest K
    K_indices = np.argsort(distances)[:K]
    K_nearest_labels = [y_train[i] for i in K_indices]
    # majority vote
    pred = max(K_nearest_labels, key = K_nearest_labels.count)
    return pred

# for whole set
def predict(X, K):
    y_pred = [_predict(x, K) for x in X]
    return y_pred
```

```python
for i in range(1,6):
    y_pred = predict(test, i)
    acc = accuracy_score(y_test, y_pred)
    print(f"Accuracy of test-set {acc}")
```

6. (20%) Plot the **1) best projection line** on the **training data** and show the slope and intercept on the title *(you can choose any value of* ***intercept*** *for better visualization)*
   2) colorize the data with each class **3)** project all data points on your projection line. Your result should look like the below image (This image is for reference, not the answer)

```python
# projection line: ax+by-+c=0
# the point we want to project: (px, py)
# projection point on line: (px - a*(a*px+b*py+c)/(a**2+b**2), py - b*(a*px+b*py+c)/(a**2+b**2))

x = x_train[i][0]
y = x_train[i][1]
a = W[1][0]
b = -W[0][0]
c = -15*W[0][0]

# point after projection
p_x = x - a*(a*x+b*y+c)/(a**2+b**2)
p_y = y - b*(a*x+b*y+c)/(a**2+b**2)
projection = np.array([p_x, p_y])
```