Assignment 2

General information

This assignment has two problems:

- 1: Linear Discriminat Analysis
- 2: Quadratic Discriminat Analysis

Utilize the designated cells within this notebook to complete the exercises. As for the Python exercises:

- Refrain from altering the provided code; simply fill in the missing portions as indicated.
- Do not use any additional libraries beyond those already included in the code (e.g., NumPy library).
- Make sure that the output of all code cells is visible in your submitted notebook. **The evaluator is NOT expected to execute your code before grading your submission.**
- Some problems include automatic checks (*assertions*) to verify basic aspects of your solution. However, passing these assertions does not guarantee that your solution is entirely correct. The final evaluation will always be determined by the evaluator.
- Avoid the use of for loops by using NumPy vectorized operations whenever possible.

Please identify the authors of this assignment in the cell below.

Identification

- **Name: Liu Cong**
- **Student Number:**
- **Name: Ulloa Ferrer Leonardo**
- **Student Number:**
- **Name:**
- **Student Number:**

Note: This work is to be done in group of up to **3** elements. Use this notebook to answer all the questions. At the end of the work, you should **upload** the **notebook** and a **pdf file** with a printout of the notebook with all the results in the **moodle** platform. To generate the pdf file we have first to covert the notebook to html using

the command `!jupyter nbconvert --to html "ML_project2.ipynb"`, then open the html file and printout to PDF.

Introduction

For this assignment, we will consider the task of predicting the quality of red wine (rated on an integer scale from 0 to 5) based on 11 chemical properties, such as pH, alcohol content, residual sugar, etc. This dataset was originally published in:

Cortez, Paulo, et al. "Modeling wine preferences by data mining from physicochemical properties." Decision support systems 47.4 (2009): 547-553.

Since quality is a discrete and ordinal attribute, this problem could be framed as either a regression or classification task. For this assignment, we will treat it as a classification problem. Specifically, we have a **6-class classification dataset** ($y \in \{0, 1, \ldots, 5\}$) where each **feature vector is 11-dimensional** ($\mathbf{x} \in \mathbb{R}^{11}$).

In the cell below, we load and prepare the data for you.

In [1]:
```
import pandas as pd

train_data = pd.read_csv("winequality_train.csv")
print("A training example:")
print(train_data.iloc[0])
print()
test_data = pd.read_csv("winequality_test.csv")
print("A test example:")
print(test_data.iloc[0])
print()

X_train = train_data.iloc[:, :-1].values
y_train = train_data.iloc[:, -1].values
X_test = test_data.iloc[:, :-1].values
y_test = test_data.iloc[:, -1].values
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
A training example:
fixed acidity            8.60000
volatile acidity         0.22000
citric acid              0.36000
residual sugar           1.90000
chlorides                0.06400
free sulfur dioxide     53.00000
total sulfur dioxide    77.00000
density                  0.99604
pH                       3.47000
sulphates                0.87000
alcohol                 11.00000
quality                  4.00000
Name: 0, dtype: float64

A test example:
fixed acidity            7.7000
volatile acidity         0.5600
citric acid              0.0800
residual sugar           2.5000
chlorides                0.1140
free sulfur dioxide     14.0000
total sulfur dioxide    46.0000
density                  0.9971
pH                       3.2400
sulphates                0.6600
alcohol                  9.6000
quality                  3.0000
Name: 0, dtype: float64

X_train shape: (1119, 11)
X_test shape: (480, 11)
y_train shape: (1119,)
y_test shape: (480,)
```

An important **preprocessing step** for multi-dimensional continuous data is to ensure all features have **zero mean** and **unit variance**. In the code below, we transform the data to achieve this.

When normalizing the test data, we reuse the mean and standard deviation from the training data, rather than recalculating them. This ensures a fair evaluation, since we want to classify new examples without assuming any prior knowledge about their distribution beyond what was learned during the training phase.

In [2]:
```
print("Training data statistics before normalization:")
print("  * mean:", X_train.mean(axis=0))
print("  * std:", X_train.std(axis=0))
print()
print("Test data statistics before normalization:")
print("  * mean:", X_test.mean(axis=0))
print("  * std:", X_test.std(axis=0))
print()

train_mean = X_train.mean(axis=0)
train_std = X_train.std(axis=0)
X_train = (X_train - train_mean) / train_std
X_test = (X_test - train_mean) / train_std
```

```
print("Training data statistics after normalization:")
print("  * mean:", X_train.mean(axis=0))
print("  * std:", X_train.std(axis=0))
print()
print("Test data statistics after normalization:")
print("  * mean:", X_test.mean(axis=0))
print("  * std:", X_test.std(axis=0))
```

```
Training data statistics before normalization:
  * mean: [ 8.30956211  0.53313226  0.27025022  2.54830206  0.08771135 15.
9204647
 46.96648794  0.996778    3.31427167  0.65882038 10.41733691]
  * std: [1.71313271e+00 1.81940308e-01 1.95404143e-01 1.42709225e+00
 4.71218617e-02 1.02685749e+01 3.30219278e+01 1.83946303e-03
 1.53911124e-01 1.72165005e-01 1.05927763e+00]

Test data statistics before normalization:
  * mean: [ 8.343125    0.5154375   0.27266667  2.51666667  0.08689583 15.
76875
 45.30520833  0.99667367  3.30375     0.65658333 10.43614583]
  * std: [1.80263540e+00 1.71324851e-01 1.93172994e-01 1.36730930e+00
 4.68790285e-02 1.08825004e+01 3.25340058e+01 1.99070295e-03
 1.55079617e-01 1.62948079e-01 1.07920724e+00]

Training data statistics after normalization:
  * mean: [ 5.42907988e-16 -2.34148913e-16  1.65094827e-16 -6.66729109e-17
  3.41301806e-16  7.93725129e-18  1.58745026e-17 -4.02418641e-14
 -8.25474134e-16 -4.42898622e-16  1.17471319e-16]
  * std: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

Test data statistics after normalization:
  * mean: [ 0.01959153 -0.09725586  0.01236639 -0.02216773 -0.01730653 -0.
01477466
 -0.05030838 -0.05671848 -0.06836199 -0.01299359  0.01775637]
  * std: [1.05224505 0.94165418 0.98858187 0.95810856 0.9948467  1.0597868
3
 0.9852243  1.0822196  1.007592   0.94646458 1.01881434]
```

Problem 1: Generative Classifiers (Linear Discriminant Analysis)

Consider a classifier that predicts the class label $\hat{y}$ for an observed feature vector $\mathbf{x}$ using the *maximum a posteriori* (MAP) classification rule:

$$\hat{y} = \arg\max_y p(y \mid \mathbf{x}).$$

Assume the following:

1. The distribution of $\mathbf{x} \mid y$ is Gaussian.
2. The covariance matrix is shared by all the classes.

In the cell below, you have the skeleton of a `LDAClassifier` class that you will complete throughout this exercise to implement this classifier.

In [3]:
```
import numpy as np
from scipy.stats import multivariate_normal
```

```python
def pda_multivariate_normal(mean: np.ndarray, cov: np.ndarray, x: np.ndar
    # I thought that why shouldn't use multivariate normal and made this,
    # the multivariate normal in the template, but i had already done thi
    F = mean.shape[0]
    temp = x - mean
    det_cov = np.linalg.det(cov)
    inv_cov = np.linalg.inv(cov)
    diagonal = np.einsum('ni,ij,nj->n', temp, inv_cov, temp)
    prefactor = 1 / np.sqrt((2 * np.pi)**F * det_cov)
    return prefactor * np.exp(-0.5 * diagonal)

class LDAClassifier:
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """
        Estimate the parameters of the classifier from the training data.

        Parameters:
        X: np.ndarray of shape (num_examples, num_features)
            The input data (features).
        y: np.ndarray of shape (num_examples,)
            The corresponding labels (target values).
        """
        # ToDo: Exercise 1.2
        # **Replace `pass` with your code**
        num_examples, num_features = X.shape
        self.classes = np.unique(y)
        num_classes = len(self.classes)

        self.means = np.zeros((num_classes, num_features))
        self.priors = np.zeros(num_classes)

        for idx, cls in enumerate(self.classes):
            X_cls = X[y == cls]
            self.means[idx] = X_cls.mean(axis=0)
            self.priors[idx] = X_cls.shape[0] / X.shape[0]

        cov_matrix = np.zeros((num_features, num_features))
        for idx, cls in enumerate(self.classes):
            X_cls = X[y == cls]
            cov_matrix += np.dot((X_cls - self.means[idx]).T, (X_cls - se
        self.covariance = cov_matrix / (X.shape[0] - num_classes)

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """
        Predict the class probabilities for the given input data.

        Parameters:
        X: np.ndarray of shape (num_examples, num_features)
            The input data (features).

        Returns:
        np.ndarray of shape (num_examples, num_classes)
            The predicted probabilities for each class.
        """
        # ToDo: Exercise 1.3
        # **Replace `pass` with your code**
        num_samples = X.shape[0]
        num_classes = len(self.classes)
```

```
        probs = np.zeros((num_samples, num_classes))



        for idx, cls in enumerate(self.classes):
            mean = self.means[idx]
            cov = self.covariance
            likelihood = pda_multivariate_normal(mean,cov, X)
            probs[:, idx] = likelihood * self.priors[idx]

        probs_sum = probs.sum(axis=1, keepdims=True)
        probs = probs / probs_sum
        return probs

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predict the class labels for the input data using the estimated m

        Parameters:
        X: np.ndarray of shape (num_examples, num_features)
            The input data (features) for which predictions are needed.

        Returns:
        np.ndarray of shape (num_examples,)
            The predicted class labels.
        """
        # ToDo: Exercise 1.4
        # **Replace `pass` with your code**
        probs = self.predict_proba(X)


        predictions = np.argmax(probs, axis=1)
        return self.classes[predictions]
```

1.1

Based on the assumptions given before, **identify all the parameters** in the model and **write down the expressions for their maximum likelihood estimates**. You do not need to show the derivation of the expressions, just the final result.

Notation:

- N_i: sample size in class c_i
- N: sample size
- K: number of classes

Prior of class c_i:

$$\hat{P}(c_i) = \frac{N_i}{N}$$

Mean of class c_i:

$$\hat{\mu}_i = \frac{1}{N_i - 1} \sum_{i:y_i=y} X_i$$

Covariance Matrix of c_i:

$$\hat{\sum}_i = \frac{1}{N_i - 1} \sum_{i=1}^{N} (X_i - \hat{\mu}_{y_i})(X_i - \hat{\mu}_{y_i})^T$$

Pooled covariance matrix c_i:

$$\hat{\sum} = \frac{1}{N-K} \sum_{i=1}^{K} (N_i - 1)\hat{\sum}_i$$

$$= \frac{1}{N-K} \sum_{i=1}^{K} (X_i - \hat{\mu}_{y_i})(X_i - \hat{\mu}_{y_i})^T$$

1.2

In this exercise, you should write the code for the `fit` method of this classifier. This method should **estimate all model parameters** using the data provided as arguments. Specifically:

- `X` : A NumPy array of shape `(num_examples, num_features)` containing the training input examples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$.
- `y` : A NumPy array of shape `(num_examples,)` containing the corresponding labels $y_1, y_2, \ldots, y_n$.

The model parameters should be saved as class attributes, so they can be used later during prediction.

After implementing the function, run the cell below to make sure that all the assertions pass and that the model parameter names and values are printed.

In [4]:
```
# instantiate the classifier and train it
lda_clf = LDAClassifier()
lda_clf.fit(X_train, y_train)

assert lda_clf.__dict__.keys(), ("`fit` method not implemented yet or not
"creating the class attributes.")
print("Classifier attributes:")
for attr, value in lda_clf.__dict__.items():
    shape = f", shape {value.shape}" if hasattr(value, "shape") else ""
    print(f"  * {attr}{shape}: {value}")
```

```
Classifier attributes:
  * classes, shape (6,): [0 1 2 3 4 5]
  * means, shape (6, 11): [[ 1.37106912e-01  2.00725764e+00 -4.22060885e-0
1  1.25763224e-01
   8.43198752e-01 -4.25074483e-01 -6.24831922e-01  4.61367497e-01
   4.34272965e-01 -5.54625151e-01 -4.35939868e-01]
 [-2.69879785e-01  6.80509915e-01 -3.01228692e-01  5.76371290e-02
   1.74129734e-01 -2.46536238e-01 -2.00871210e-01 -1.57623531e-01
   2.66427180e-01 -2.57752328e-01 -2.01241142e-01]
 [-6.28732712e-02  2.80444275e-01 -1.28167508e-01  1.39499189e-02
   8.84356273e-02  1.03727345e-01  3.17583390e-01  2.08867168e-01
  -6.18446834e-02 -2.18073674e-01 -5.01012554e-01]
 [ 8.15345524e-04 -1.75140616e-01  5.84670637e-03 -6.61630507e-02
  -4.65407032e-02 -2.28261122e-02 -1.98174787e-01 -8.46028330e-02
   7.80116953e-02  1.06442906e-01  2.14973258e-01]
 [ 2.43136358e-01 -6.90115339e-01  4.72653602e-01  1.26863397e-01
  -2.32386401e-01 -1.77496745e-01 -3.57332901e-01 -4.13927324e-01
  -8.47228373e-02  4.78677468e-01  1.04271341e+00]
 [ 4.27348421e-01 -5.76007202e-01  7.62094604e-01  1.23816764e-01
  -3.65251898e-01 -2.68177237e-01 -4.48181625e-01 -4.84107698e-01
  -4.93390835e-01  6.31252702e-01  1.38395863e+00]]
  * priors, shape (6,): [0.0080429  0.03217158 0.43431635 0.39142091 0.123
3244  0.01072386]
  * covariance, shape (11, 11): [[ 0.99185791 -0.2121593   0.6407643   0.1
190726   0.11123889 -0.13836573
  -0.08337088  0.6787589  -0.67692057  0.13942941 -0.0976629 ]
 [-0.2121593   0.84878973 -0.47931437 -0.00503167  0.01763438 -0.0356702
   0.00119635 -0.05551758  0.21245081 -0.17773231 -0.02534906]
 [ 0.6407643  -0.47931437  0.95986782  0.16788067  0.24222922 -0.04259647
   0.08889401  0.41062754 -0.52664782  0.25618725  0.01047294]
 [ 0.1190726  -0.00503167  0.16788067  1.001187    0.0647507   0.15263664
   0.16655545  0.38013492 -0.09036379  0.00879145  0.04309229]
 [ 0.11123889  0.01763438  0.24222922  0.0647507   0.98625924 -0.00920229
   0.01441464  0.16619225 -0.26742823  0.39440265 -0.13816211]
 [-0.13836573 -0.0356702  -0.04259647  0.15263664 -0.00920229  0.99237891
   0.64314801 -0.04786564  0.05826412  0.06942813 -0.04390487]
 [-0.08337088  0.00119635  0.08889401  0.16655545  0.01441464  0.64314801
   0.92343509  0.01647462 -0.06328205  0.09085758 -0.09634875]
 [ 0.6787589  -0.05551758  0.41062754  0.38013492  0.16619225 -0.04786564
   0.01647462  0.95722923 -0.32980707  0.18221407 -0.36132685]
 [-0.67692057  0.21245081 -0.52664782 -0.09036379 -0.26742823  0.05826412
  -0.06328205 -0.32980707  0.99399017 -0.18901265  0.19996288]
 [ 0.13942941 -0.17773231  0.25618725  0.00879145  0.39440265  0.06942813
   0.09085758  0.18221407 -0.18901265  0.94282387 -0.01806321]
 [-0.0976629  -0.02534906  0.01047294  0.04309229 -0.13816211 -0.04390487
  -0.09634875 -0.36132685  0.19996288 -0.01806321  0.71929272]]
```

1.3

Implement the `predict_proba` method of the `LDAClassifier` class. This method should use the model parameters estimated during the `fit` method to **predict the probabilities of each class** for new input data. Specifically, the input is:

- `X`: A NumPy array of shape `(num_examples, num_features)` containing the input examples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ for which predictions are needed.

The method should return a NumPy array of shape `(num_examples, num_classes)` containing the predicted probabilities

$p(y = 0 \mid \mathbf{x}_i), p(y = 1 \mid \mathbf{x}_i), \ldots, p(y = C - 1 \mid \mathbf{x}_i)$, for $i \in \{1, 2, \ldots, n\}$.

After implementing the function, run the cell below to make sure that all the assertions pass.

In [5]:
```
# instantiate the classifier
# (we need to do this again because you probably changed the code of LDAC
# after you ran the previous cell)
lda_clf = LDAClassifier()
lda_clf.fit(X_train, y_train)

# predict the class probabilities for the training data
probs = lda_clf.predict_proba(X_train)

assert probs is not None, ("`predict_proba` method not implemented yet or
"not returning a valid array.")
assert probs.shape == (len(X_train), 6), ("`predict_proba` output should
"have shape (num_examples, num_classes).")
assert np.all((probs >= 0) * (probs <= 1)), ("`predict_proba` output shou
"probabilities.")
assert np.allclose(probs.sum(axis=1), 1), ("`predict_proba` output should
"(proper) probability distribution.")
print("Your `predict_proba` looks good! 🚀")
```

```
Your `predict_proba` looks good! 🚀
```

1.4

Implement the `predict` method of the `LDAClassifier` class. This method should **predict the class labels** for new input data, by applying the MAP rule. Specifically, the input is:

- `X`: A NumPy array of shape `(num_examples, num_features)` containing the input examples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ for which predictions are needed.

The method should return a NumPy array of shape `(num_examples,)` containing the predicted class labels $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$.

After implementing the function, run the cell below to make sure that all the assertions pass and that a summary of the model predictions is printed.

In [6]:
```
# instantiate the classifier
# (we need to do this again because you probably changed the code of LDAC
# after you ran the previous cell)
lda_clf = LDAClassifier()
lda_clf.fit(X_train, y_train)

# predict the class labels for the training data
y_train_pred = lda_clf.predict(X_train)
assert y_train_pred is not None, ("`predict` method not implemented yet o
"any predictions.")
assert len(y_train_pred) == len(X_train), ("You should have one predicted
"label for each input example.")
print("Your classifier predicted:")
for j in range(6):
    print(f"  * {(y_train_pred == j).sum()} training examples in class {j
```

```
print()

# predict the class labels for the test data
y_test_pred = lda_clf.predict(X_test)
assert y_test_pred is not None, ("`predict` method not implemented yet or
"any predictions.")
assert len(y_test_pred) == len(X_test), ("You should have one predicted "
"label for each input example.")
print("Your classifier predicted:")
for j in range(6):
    print(f"  * {(y_test_pred == j).sum()} test examples in class {j}.")
```

```
Your classifier predicted:
  * 8 training examples in class 0.
  * 1 training examples in class 1.
  * 547 training examples in class 2.
  * 453 training examples in class 3.
  * 109 training examples in class 4.
  * 1 training examples in class 5.

Your classifier predicted:
  * 3 test examples in class 0.
  * 0 test examples in class 1.
  * 227 test examples in class 2.
  * 192 test examples in class 3.
  * 58 test examples in class 4.
  * 0 test examples in class 5.
```

1.5

Complete the code of the function `compute_accuracy`, which **computes the accuracy** of a given set of predictions when compared to the ground-truth labels. This function will be used to compute the training and test accuracies of our `LDAClassifier`.

After implementing the function, run the cell below to make sure that all the assertions pass and that the values of the training and test accuracies are reasonable.

In [7]:
```
def compute_accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Computes the accuracy of the classifier.

    Parameters:
    y_true: np.ndarray of shape (num_examples,)
        The true class labels.
    y_pred: np.ndarray of shape (num_examples,)
        The predicted class labels.

    Returns:
    float
        The accuracy of the classifier.
    """
    # ToDo:
    # **Replace `pass` with your code**
    num_correct = np.sum(y_true == y_pred)
```

```
    accuracy = num_correct / len(y_true)

    return accuracy


# compute the training accuracy
train_accuracy = compute_accuracy(y_train, y_train_pred)
assert train_accuracy is not None, ("`compute_accuracy` not implemented y
"not returning a valid float.")
print(f"Training accuracy: {train_accuracy:.3f}")

# compute the test accuracy
test_accuracy = compute_accuracy(y_test, y_test_pred)
assert test_accuracy is not None, ("`compute_accuracy` not implemented ye
"not returning a valid float.")
print(f"Test accuracy: {test_accuracy:.3f}")
```

```
Training accuracy: 0.624
Test accuracy: 0.565
```

1.6

Explain why normalizing features to have zero mean and unit variance is an important preprocessing step **for this classifier**.

LDA assumes that the features within each category follow a Gaussian distribution and that all categories share the same covariance matrix. Normalizing the training data helps align the features to this assumption by standardizing their scales and variances, thereby reducing errors.

Problem 2: Quadratic Discriminant Analysis (QDA)

You should now repeat the analysis performed on the previous exercise (items 1.2 to 1.5), but now using the QDA model. At the end, compare the performance of both methods and define which one would you choose to deploy. In the cell below, you have the skeleton of a `QDAClassifier` class that you will complete throughout this exercise to implement this classifier.

In [8]:
```
import numpy as np
from scipy.stats import multivariate_normal

class QDAClassifier:
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """
        Estimate the parameters of the classifier from the training data.

        Parameters:
        X: np.ndarray of shape (num_examples, num_features)
            The input data (features).
        y: np.ndarray of shape (num_examples,)
            The corresponding labels (target values).
        """
        self.classes = np.unique(y)
        _, self.num_features = X.shape
        self.num_classes = len(self.classes)
```

```
        self.means, self.covs = [], []
        self.priors = np.zeros(self.num_classes)
        epsilon = 1e-5  # Small regularization term
        for c in range(self.num_classes):
            mask = y == c
            Xc = X[mask]
            mean = np.mean(Xc,axis = 0)
            cov = np.cov(Xc.T)
            self.means.append(mean)
            self.covs.append(cov + epsilon*np.eye(self.num_features))
            self.priors[c] = np.sum(mask)/len(y)


    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """
        Predict the class probabilities for the given input data.

        Parameters:
        X: np.ndarray of shape (num_examples, num_features)
            The input data (features).

        Returns:
        np.ndarray of shape (num_examples, num_classes)
            The predicted probabilities for each class.
        """
        # ToDo:
        num_samples = X.shape[0]
        num_classes = self.num_classes
        likelihoods = np.full((num_samples,num_classes),np.nan)
        for c in range(num_classes):
            likelihoods[:,c] = multivariate_normal.pdf(X,self.means[c],se
        posteriors = (self.priors.reshape(1,self.num_classes)*likelihoods
        posteriors = posteriors/np.sum(posteriors,axis=1).reshape(num_sam
        return posteriors

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predict the class labels for the input data using the estimated m

        Parameters:
        X: np.ndarray of shape (num_examples, num_features)
            The input data (features) for which predictions are needed.

        Returns:
        np.ndarray of shape (num_examples,)
            The predicted class labels.
        """
        probs = self.predict_proba(X)
        predictions = np.argmax(probs, axis=1)
        return self.classes[predictions]
```

2.1

What is the difference between the estimated parameters of LDA and QDA?

The primary difference between the estimated parameters in LDA and QDA lies in the treatment of the covariance matrix. In LDA, the covariance matrices of all classes are

assumed to be the same, and a pooled covariance matrix is estimated. This assumption simplifies computations and results in linear decision boundaries. In contrast, QDA allows each class to have its own covariance matrix, leading to more flexible decision boundaries with quadratic shapes.

2.2

Write your fit method:

In [9]:
```
# instantiate the classifier and train it
qda_clf = QDAClassifier()
qda_clf.fit(X_train, y_train)

assert qda_clf.__dict__.keys(), ("`fit` method not implemented yet or not
"creating the class attributes.")
print("Classifier attributes:")
for attr, value in qda_clf.__dict__.items():
    shape = f", shape {value.shape}" if hasattr(value, "shape") else ""
    print(f"  * {attr}{shape}: {value}")
```

```
Classifier attributes:
  * classes, shape (6,): [0 1 2 3 4 5]
  * num_features: 11
  * num_classes: 6
  * means: [array([ 0.13710691,  2.00725764, -0.42206088,  0.12576322,  0.
84319875,
       -0.42507448, -0.62483192,  0.4613675 ,  0.43427296, -0.55462515,
       -0.43593987]), array([-0.26987979,  0.68050992, -0.30122869,  0.057
63713,  0.17412973,
       -0.24653624, -0.20087121, -0.15762353,  0.26642718, -0.25775233,
       -0.20124114]), array([-0.06287327,  0.28044427, -0.12816751,  0.013
94992,  0.08843563,
        0.10372735,  0.31758339,  0.20886717, -0.06184468, -0.21807367,
       -0.50101255]), array([ 0.00081535, -0.17514062,  0.00584671, -0.066
16305, -0.0465407 ,
       -0.02282611, -0.19817479, -0.08460283,  0.0780117 ,  0.10644291,
        0.21497326]), array([ 0.24313636, -0.69011534,  0.4726536 ,  0.126
8634 , -0.2323864 ,
       -0.17749675, -0.3573329 , -0.41392732, -0.08472284,  0.47867747,
        1.04271341]), array([ 0.42734842, -0.5760072 ,  0.7620946 ,  0.123
81676, -0.3652519 ,
       -0.26817724, -0.44818163, -0.4841077 , -0.49339084,  0.6312527 ,
        1.38395863])]
  * covs: [array([[ 1.07171929, -1.3163542 ,  1.33153641, -0.36895226, -0.
08482053,
         0.24072642,  0.22840052,  0.91195459, -0.50557691,  0.45164323,
        -0.52848222],
       [-1.3163542 ,  3.66422183, -1.89347646,  0.4855978 , -0.51683804,
        -0.96390542, -0.59982327, -1.39873695,  1.20197012, -1.11696382,
         1.13427852],
       [ 1.33153641, -1.89347646,  1.76833189, -0.46660754,  0.01613969,
         0.31591424,  0.3465    ,  1.13866195, -0.81454122,  0.58471483,
        -0.84298004],
       [-0.36895226,  0.4855978 , -0.46660754,  1.03762291, -0.79373348,
        -0.59487102, -0.27634639,  0.17296276,  0.44060923, -0.21919083,
         0.32381963],
       [-0.08482053, -0.51683804,  0.01613969, -0.79373348,  2.09924038,
         0.21384109, -0.09409481, -0.67837522, -0.42147844,  0.11458322,
        -0.34464052],
       [ 0.24072642, -0.96390542,  0.31591424, -0.59487102,  0.21384109,
         0.9842115 ,  0.40930853, -0.02894152, -0.18709535,  0.55598317,
        -0.13075174],
       [ 0.22840052, -0.59982327,  0.3465    , -0.27634639, -0.09409481,
         0.40930853,  0.27099961,  0.16125417, -0.22389172,  0.22448558,
        -0.19892667],
       [ 0.91195459, -1.39873695,  1.13866195,  0.17296276, -0.67837522,
        -0.02894152,  0.16125417,  1.24442271, -0.46340314,  0.34970979,
        -0.46435243],
       [-0.50557691,  1.20197012, -0.81454122,  0.44060923, -0.42147844,
        -0.18709535, -0.22389172, -0.46340314,  0.8500426 , -0.21243654,
         0.6919117 ],
       [ 0.45164323, -1.11696382,  0.58471483, -0.21919083,  0.11458322,
         0.55598317,  0.22448558,  0.34970979, -0.21243654,  0.54824102,
        -0.25611776],
       [-0.52848222,  1.13427852, -0.84298004,  0.32381963, -0.34464052,
        -0.13075174, -0.19892667, -0.46435243,  0.6919117 , -0.25611776,
         0.67089361]]), array([[ 0.97499895, -0.2893284 ,  0.54715263,  0.
01330002,  0.32860422,
         0.26122927,  0.09527018,  0.6781476 , -0.79590475,  0.35265535,
        -0.34546894],
```

```
           [-0.2893284 ,  1.55472736, -0.66224908, -0.55027081, -0.05373508,
            -0.07282869, -0.16767271, -0.05401988,  0.44881544, -0.33877632,
             0.0445428 ],
           [ 0.54715263, -0.66224908,  1.25729374,  0.39389718,  1.34365534,
             0.39002178,  0.36859142,  0.50064545, -0.82808942,  1.0660033 ,
            -0.14379193],
           [ 0.01330002, -0.55027081,  0.39389718,  2.00470872, -0.01517145,
            -0.21452212,  0.42623921,  0.1077423 , -0.33493022,  0.0410626 ,
             0.45607359],
           [ 0.32860422, -0.05373508,  1.34365534, -0.01517145,  3.77479345,
             0.57140964,  0.20876974,  0.63534779, -1.29992844,  2.63382476,
            -0.34297499],
           [ 0.26122927, -0.07282869,  0.39002178, -0.21452212,  0.57140964,
             0.91385117,  0.4950532 ,  0.20176459, -0.14704491,  0.66774994,
            -0.26650153],
           [ 0.09527018, -0.16767271,  0.36859142,  0.42623921,  0.20876974,
             0.4950532 ,  0.85470481,  0.23377954, -0.14623641,  0.4978657 ,
            -0.06052544],
           [ 0.6781476 , -0.05401988,  0.50064545,  0.1077423 ,  0.63534779,
             0.20176459,  0.23377954,  0.81381333, -0.67413881,  0.59932306,
            -0.28139871],
           [-0.79590475,  0.44881544, -0.82808942, -0.33493022, -1.29992844,
            -0.14704491, -0.14623641, -0.67413881,  1.57122431, -1.17134442,
             0.60744478],
           [ 0.35265535, -0.33877632,  1.0660033 ,  0.0410626 ,  2.63382476,
             0.66774994,  0.4978657 ,  0.59932306, -1.17134442,  2.70582663,
            -0.2763082 ],
           [-0.34546894,  0.0445428 , -0.14379193,  0.45607359, -0.34297499,
            -0.26650153, -0.06052544, -0.28139871,  0.60744478, -0.2763082 ,
             0.79476306]]), array([[ 8.12077498e-01, -1.47092297e-01,  5.02073
581e-01,
             1.23952295e-01,  8.09409547e-02, -1.00867645e-01,
            -5.32533888e-02,  4.92283383e-01, -5.59135572e-01,
             7.89592317e-02,  1.26994275e-02],
           [-1.47092297e-01,  8.84311126e-01, -4.12556003e-01,
            -2.59178332e-02,  6.18559785e-03, -3.71377780e-02,
             9.44219675e-03, -9.65177990e-02,  1.33728109e-01,
            -1.70736139e-01, -1.28405992e-02],
           [ 5.02073581e-01, -4.12556003e-01,  8.47400668e-01,
             1.44346513e-01,  2.65544975e-01,  3.39372062e-02,
             1.93825843e-01,  3.24479172e-01, -4.17221644e-01,
             2.49399960e-01,  3.96831562e-04],
           [ 1.23952295e-01, -2.59178332e-02,  1.44346513e-01,
             1.07017378e+00,  5.73027551e-02,  3.11768320e-01,
             2.21694811e-01,  4.10970946e-01, -8.45417577e-02,
             4.58384282e-02,  3.63809643e-02],
           [ 8.09409547e-02,  6.18559785e-03,  2.65544975e-01,
             5.73027551e-02,  1.19024554e+00,  3.71662898e-02,
             6.09448642e-02,  9.52245173e-02, -2.90896766e-01,
             5.01714726e-01, -1.15506942e-01],
           [-1.00867645e-01, -3.71377780e-02,  3.39372062e-02,
             3.11768320e-01,  3.71662898e-02,  1.10761113e+00,
             8.31059520e-01,  6.34434329e-02, -3.18802076e-02,
             6.02838181e-02, -1.04975085e-01],
           [-5.32533888e-02,  9.44219675e-03,  1.93825843e-01,
             2.21694811e-01,  6.09448642e-02,  8.31059520e-01,
             1.29826462e+00,  8.08751160e-02, -1.51205038e-01,
             1.41507704e-01, -1.62175083e-01],
           [ 4.92283383e-01, -9.65177990e-02,  3.24479172e-01,
             4.10970946e-01,  9.52245173e-02,  6.34434329e-02,
```

```
          8.08751160e-02,  6.96870389e-01, -2.06747824e-01,
          1.31756938e-01, -1.60525310e-01],
        [-5.59135572e-01,  1.33728109e-01, -4.17221644e-01,
         -8.45417577e-02, -2.90896766e-01, -3.18802076e-02,
         -1.51205038e-01, -2.06747824e-01,  9.61480312e-01,
         -2.04569889e-01,  1.47206679e-01],
        [ 7.89592317e-02, -1.70736139e-01,  2.49399960e-01,
          4.58384282e-02,  5.01714726e-01,  6.02838181e-02,
          1.41507704e-01,  1.31756938e-01, -2.04569889e-01,
          1.02785032e+00,  1.07947461e-02],
        [ 1.26994275e-02, -1.28405992e-02,  3.96831562e-04,
          3.63809643e-02, -1.15506942e-01, -1.04975085e-01,
         -1.62175083e-01, -1.60525310e-01,  1.47206679e-01,
          1.07947461e-02,  4.65089806e-01]]), array([[ 1.08798495, -0.26100
291,  0.70179306,  0.08892589,  0.12116338,
         -0.2222443 , -0.11003892,  0.7535686 , -0.74505515,  0.18781755,
         -0.10642223],
        [-0.26100291,  0.76636254, -0.53444123,  0.02933805,  0.03812893,
         -0.01055601,  0.01765207, -0.02264599,  0.23798963, -0.16325889,
         -0.08358377],
        [ 0.70179306, -0.53444123,  1.01074818,  0.14155089,  0.15722005,
         -0.16876716, -0.02717844,  0.40967799, -0.54504171,  0.2291843 ,
          0.06892701],
        [ 0.08892589,  0.02933805,  0.14155089,  0.86973065,  0.08325331,
          0.07053664,  0.08956018,  0.35406783, -0.06158433, -0.01099153,
         -0.0065923 ],
        [ 0.12116338,  0.03812893,  0.15722005,  0.08325331,  0.69176307,
         -0.06039987,  0.00671051,  0.19161437, -0.20444878,  0.20485288,
         -0.14594843],
        [-0.2222443 , -0.01055601, -0.16876716,  0.07053664, -0.06039987,
          0.8673509 ,  0.4538088 , -0.16755579,  0.17967461,  0.0378447 ,
          0.03994603],
        [-0.11003892,  0.01765207, -0.02717844,  0.08956018,  0.00671051,
          0.4538088 ,  0.5396134 , -0.00685449,  0.012444  ,  0.03871211,
         -0.08659353],
        [ 0.7535686 , -0.02264599,  0.40967799,  0.35406783,  0.19161437,
         -0.16755579, -0.00685449,  1.09804006, -0.3734154 ,  0.19902857,
         -0.50498553],
        [-0.74505515,  0.23798963, -0.54504171, -0.06158433, -0.20444878,
          0.17967461,  0.012444  , -0.3734154 ,  1.00730342, -0.14286688,
          0.18635779],
        [ 0.18781755, -0.16325889,  0.2291843 , -0.01099153,  0.20485288,
          0.0378447 ,  0.03871211,  0.19902857, -0.14286688,  0.83479051,
         -0.02842073],
        [-0.10642223, -0.08358377,  0.06892701, -0.0065923 , -0.14594843,
          0.03994603, -0.08659353, -0.50498553,  0.18635779, -0.02842073,
          0.92970694]]), array([[ 1.29163832, -0.18071514,  0.88520613,  0.
2176656 ,  0.13780605,
         -0.11956392, -0.14852483,  1.06037424, -0.82002012,  0.1617187 ,
         -0.35642088],
        [-0.18071514,  0.63972428, -0.40575544,  0.07521008,  0.05298185,
         -0.04753702, -0.02912672,  0.09127584,  0.24364563, -0.16493332,
         -0.01759916],
        [ 0.88520613, -0.40575544,  1.06229591,  0.29443248,  0.16788838,
         -0.01751666,  0.03574624,  0.61237869, -0.71989277,  0.17469525,
         -0.02701377],
        [ 0.2176656 ,  0.07521008,  0.29443248,  0.90175476,  0.09834267,
          0.02739744,  0.21280802,  0.402695  , -0.17335472, -0.05317811,
          0.06466954],
        [ 0.13780605,  0.05298185,  0.16788838,  0.09834267,  0.5012622 ,
```

```
        -0.1763031 , -0.16325913,  0.26744373, -0.12103357,  0.10076133,
        -0.14243328],
       [-0.11956392, -0.04753702, -0.01751666,  0.02739744, -0.1763031 ,
         0.96492377,  0.63697562, -0.07892736,  0.06178776,  0.05296584,
        -0.06047014],
       [-0.14852483, -0.02912672,  0.03574624,  0.21280802, -0.16325913,
         0.63697562,  0.90401936, -0.16105309,  0.01489552, -0.02149039,
         0.07607133],
       [ 1.06037424,  0.09127584,  0.61237869,  0.402695  ,  0.26744373,
        -0.07892736, -0.16105309,  1.42933366, -0.49958553,  0.1974633 ,
        -0.63881648],
       [-0.82002012,  0.24364563, -0.71989277, -0.17335472, -0.12103357,
         0.06178776,  0.01489552, -0.49958553,  0.88980523, -0.05495762,
         0.25068717],
       [ 0.1617187 , -0.16493332,  0.17469525, -0.05317811,  0.10076133,
         0.05296584, -0.02149039,  0.1974633 , -0.05495762,  0.58967154,
         0.01467363],
       [-0.35642088, -0.01759916, -0.02701377,  0.06466954, -0.14243328,
        -0.06047014,  0.07607133, -0.63881648,  0.25068717,  0.01467363,
         0.89393771]]), array([[ 1.36261832, -0.4836326 ,  1.08232147,  0.
56511445,  0.17291939,
        -0.24073359, -0.33551328,  1.00815156, -1.12715427, -0.29982833,
        -0.29114357],
       [-0.4836326 ,  0.86830149, -0.53830834, -0.07090084, -0.11621616,
        -0.02757375,  0.33596508, -0.40980365,  0.80858185, -0.02510439,
         0.57437695],
       [ 1.08232147, -0.53830834,  1.0882961 ,  0.41769125,  0.17717179,
        -0.35422365, -0.34176854,  0.91809182, -1.04532048, -0.17247235,
        -0.28906774],
       [ 0.56511445, -0.07090084,  0.41769125,  1.20177238,  0.11835565,
        -0.33034269, -0.28598783,  0.79252027, -0.06477383, -0.00379258,
         0.52590255],
       [ 0.17291939, -0.11621616,  0.17717179,  0.11835565,  0.05122766,
         0.05185418, -0.05772174,  0.14569443, -0.16069519, -0.07154856,
         0.0273191 ],
       [-0.24073359, -0.02757375, -0.35422365, -0.33034269,  0.05185418,
         1.47746179,  0.59803885, -0.62180153, -0.00268432, -0.32524662,
         0.29530578],
       [-0.33551328,  0.33596508, -0.34176854, -0.28598783, -0.05772174,
         0.59803885,  0.5810613 , -0.48189172,  0.21201929, -0.06372184,
         0.23165156],
       [ 1.00815156, -0.40980365,  0.91809182,  0.79252027,  0.14569443,
        -0.62180153, -0.48189172,  1.21127778, -0.71588618,  0.10001858,
        -0.23106328],
       [-1.12715427,  0.80858185, -1.04532048, -0.06477383, -0.16069519,
        -0.00268432,  0.21201929, -0.71588618,  1.46510337,  0.13671644,
         0.78046265],
       [-0.29982833, -0.02510439, -0.17247235, -0.00379258, -0.07154856,
        -0.32524662, -0.06372184,  0.10001858,  0.13671644,  0.56257937,
        -0.29186333],
       [-0.29114357,  0.57437695, -0.28906774,  0.52590255,  0.0273191 ,
         0.29530578,  0.23165156, -0.23106328,  0.78046265, -0.29186333,
         1.18910113]])]
  * priors, shape (6,): [0.0080429  0.03217158 0.43431635 0.39142091 0.123
3244  0.01072386]
```

2.3

Implement the predict_proba method:

In [10]:
```
# instantiate the classifier
# (we need to do this again because you probably changed the code of QDAC
# after you ran the previous cell)
qda_clf = QDAClassifier()
qda_clf.fit(X_train, y_train)

# predict the class probabilities for the training data
probs = qda_clf.predict_proba(X_train)

assert probs is not None, ("`predict_proba` method not implemented yet or
"not returning a valid array.")
assert probs.shape == (len(X_train), 6), ("`predict_proba` output should
"have shape (num_examples, num_classes).")
assert np.all((probs >= 0) * (probs <= 1)), ("`predict_proba` output shou
"probabilities.")
assert np.allclose(probs.sum(axis=1), 1), ("`predict_proba` output should
"(proper) probability distribution.")
print("Your `predict_proba` looks good! 🚀")
```

```
Your `predict_proba` looks good! 🚀
```

2.4

Write your predict method:

In [11]:
```
# instantiate the classifier
# (we need to do this again because you probably changed the code of QDAC
# after you ran the previous cell)
qda_clf = QDAClassifier()
qda_clf.fit(X_train, y_train)

# predict the class labels for the training data
y_train_pred = qda_clf.predict(X_train)
assert y_train_pred is not None, ("`predict` method not implemented yet o
"any predictions.")
assert len(y_train_pred) == len(X_train), ("You should have one predicted
"label for each input example.")
print("Your classifier predicted:")
for j in range(6):
    print(f"  * {(y_train_pred == j).sum()} training examples in class {j
print()

# predict the class labels for the test data
y_test_pred = qda_clf.predict(X_test)
assert y_test_pred is not None, ("`predict` method not implemented yet or
"any predictions.")
assert len(y_test_pred) == len(X_test), ("You should have one predicted "
"label for each input example.")
print("Your classifier predicted:")
for j in range(6):
    print(f"  * {(y_test_pred == j).sum()} test examples in class {j}.")
```

```
Your classifier predicted:
  * 9 training examples in class 0.
  * 32 training examples in class 1.
  * 421 training examples in class 2.
  * 528 training examples in class 3.
  * 116 training examples in class 4.
  * 13 training examples in class 5.

Your classifier predicted:
  * 0 test examples in class 0.
  * 10 test examples in class 1.
  * 173 test examples in class 2.
  * 225 test examples in class 3.
  * 66 test examples in class 4.
  * 6 test examples in class 5.
```

2.5

Using the compute_accuracy that you have already defined, compute the performance of the QDA on the given data. Compare the results with the ones obtained with LDA and define which one would you deploy.

In [12]:
```
# compute the training accuracy
train_accuracy = compute_accuracy(y_train, y_train_pred)
assert train_accuracy is not None, ("`compute_accuracy` not implemented y
"not returning a valid float.")
print(f"Training accuracy: {train_accuracy:.3f}")

# compute the test accuracy
test_accuracy = compute_accuracy(y_test, y_test_pred)
assert test_accuracy is not None, ("`compute_accuracy` not implemented ye
"not returning a valid float.")
print(f"Test accuracy: {test_accuracy:.3f}")
```

```
Training accuracy: 0.634
Test accuracy: 0.569
```

Taking into account the accuracy of each method, QDA performs slightly better than LDA on this dataset. However, this difference becomes almost insignificant on the test set, suggesting that some overfitting may be occurring. Despite this, QDA still demonstrates a better ability to learn from these features and would be my choice for deployment.