Statistical Learning weekly assignment 2

Gaspard Gaches (s4645251)

February 18, 2025

[64]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,␣
 ↪QuadraticDiscriminantAnalysis
from scipy.stats import norm
from statistics import fmean
```

First let's recreate the R function given in the assignment in python:

[33]:
```python
# note: after experimenting I realised np.diag() only creates a
# strided view of the original data, not a regular 2D array, so
# to make operations on it I need to add a float to it (here 0.2),
# or just '+ 0.'
p = 3

def generateData(n):
    n1 = n2 = n//2
    cov_1 = np.diag(np.repeat(1, p)) + 0.2
    cov_2 = np.diag(np.repeat(1, p)) + 0.2
    cov_2[1, 2] = cov_2[2, 1] = cov_2[1, 2] + 0.5
    x_class1 = np.random.multivariate_normal(
        mean=np.repeat(3, p),
        cov=cov_1,
        size=n1
    )
    x_class2 = np.random.multivariate_normal(
        mean=np.repeat(2, p),
        cov=cov_2,
        size=n2
    )
    y = np.repeat((1, 2), (n1, n2))
    data_set = pd.concat([pd.DataFrame(x_class1), pd.DataFrame(x_class2)])
    data_set.columns = [f'x_{i}' for i in range(1, p+1)]
    data_set['y'] = y

    return data_set
```

Since the covariance matrices are slightly different between the 2 classes, it follows the assumption that the validity (or let's say the performance) of QDA depends upon. In this case, QDA might be the best performing classifier, while LDA would have have higher bias, so it might lack a little bit of flexibility for our data set here (given that the Bayes decision boundary isn't linear).

[ ]:
```
def predictLDA(train_set, test_set):

    LDA = LinearDiscriminantAnalysis(store_covariance=True)
    LDA.fit(
        X=train_set[[f'x_{i}' for i in range(1, p+1)]],
        y=train_set['y']
    )
    # let's first take a quick look at what's inside our LDA:
    print("LDA Classes:", LDA.classes_)
    print("LDA Means:", LDA.means_)
    print("LDA Covariance matrix:", LDA.covariance_)

    y_pred = LDA.predict(X=test_set[[f'x_{i}' for i in range(1, p+1)]])

    return y_pred


def predictQDA(train_set, test_set):

    QDA = QuadraticDiscriminantAnalysis(store_covariance=True)
    QDA.fit(
        X=train_set[[f'x_{i}' for i in range(1, p+1)]],
        y=train_set['y']
    )
    # let's first take a quick look at what's inside our QDA:
    print("QDA Classes:", QDA.classes_)
    print("QDA Means:", QDA.means_)
    print("QDA Covariance matrix:", QDA.covariance_)

    y_pred = QDA.predict(X=test_set[[f'x_{i}' for i in range(1, p+1)]])

    return y_pred
```

First let's generate two small data sets and take a look at what our discriminant functions do:

[117]:
```
train = generateData(20)
test = generateData(20)
```

[118]:
```
predictLDA(train, test)
```

```
LDA Classes: [1 2]
LDA Means: [[3.22560067 2.39104093 3.38032654]
 [2.03234439 1.7851859  1.72716202]]
LDA Covariance matrix: [[ 1.13749041 -0.21471272 -0.20747685]
 [-0.21471272  0.70179146 -0.1659123 ]
 [-0.20747685 -0.1659123   0.94067546]]
```

```
c:\Users\Gaspard\miniconda3\envs\venv_stats\Lib\site-
packages\sklearn\utils\validation.py:1408: DataConversionWarning: A column-
vector y was passed when a 1d array was expected. Please change the shape of y
to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

```
[118]: array([1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
[ ]: predictQDA(train, test)
```

```
QDA Classes: [1 2]
QDA Means: [[3.22560067 2.39104093 3.38032654]
 [2.03234439 1.7851859  1.72716202]]
QDA Covariance matrix: [array([[ 0.94555935, -0.23851467,  0.08941347],
       [-0.23851467,  0.4644733 , -0.49326297],
       [ 0.08941347, -0.49326297,  1.17158881]]), array([[ 1.58219712,
-0.23862471, -0.55047315],
       [-0.23862471,  1.09506328,  0.12456897],
       [-0.55047315,  0.12456897,  0.91880109]])]
```

```
c:\Users\Gaspard\miniconda3\envs\venv_stats\Lib\site-
packages\sklearn\utils\validation.py:1408: DataConversionWarning: A column-
vector y was passed when a 1d array was expected. Please change the shape of y
to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

```
[ ]: array([1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

As we can see, the QDA function returned two covariances matrices. Also notice how the fifth elements of the `y_pred` results differ betwee the two methods, but the rest of the predicted labels are the same.

I have now removed the `store_covariance=True` parameter and commented out the print statements in the discriminant functions, to prepare for the loop we are going to be running next:

```
[ ]: def predictLDA(train_set, test_set):
         LDA = LinearDiscriminantAnalysis()
         LDA.fit(
             X=train_set[[f'x_{i}' for i in range(1, p+1)]],
             y=train_set['y']
         )
         y_pred = LDA.predict(X=test_set[[f'x_{i}' for i in range(1, p+1)]])
         return y_pred

     def predictQDA(train_set, test_set):
         QDA = QuadraticDiscriminantAnalysis()
         QDA.fit(
             X=train_set[[f'x_{i}' for i in range(1, p+1)]],
             y=train_set['y']
         )
         y_pred = QDA.predict(X=test_set[[f'x_{i}' for i in range(1, p+1)]])
         return y_pred
```

Below I manually coded a function that computes the confusion matrix and returns the accuracy, for the challenge:

[35]:
```
def classifyPreds(row):

    if row.y == 2 and row.y_hat == 2:
        return 'TP'
    elif row.y == 2 and row.y_hat == 1:
        return 'FN'
    elif row.y == 1 and row.y_hat == 2:
        return 'FP'
    elif row.y == 1 and row.y_hat == 1:
        return 'TN'


def getAccuracy(df_labels):

    pos_negs = df_labels.apply(classifyPreds, axis=1)
    cf_matrix = pos_negs.value_counts()
    return (cf_matrix['TP'] + cf_matrix['TN']) / sum(cf_matrix)
```

And now the function that is going to run the experiment 100 times and return the average accuracy over these 100 repetitions for both methods:

[ ]:
```
def discriminantAnalysis(n_train):

    classifier_acc = {'LDA': [], 'QDA': []}

    for _ in range(100):
        train = generateData(n_train)
        test = generateData(10_000)

        y_hat_LDA = predictLDA(train, test)
        y_hat_QDA = predictQDA(train, test)

        df_LDA = pd.DataFrame({'y': test['y'], 'y_hat': y_hat_LDA})
        df_QDA = pd.DataFrame({'y': test['y'], 'y_hat': y_hat_QDA})

        classifier_acc['LDA'].append(getAccuracy(df_LDA))
        classifier_acc['QDA'].append(getAccuracy(df_QDA))

    return pd.DataFrame({'LDA': [round(fmean(classifier_acc['LDA']), 4)], 'QDA':␣
 ↪[round(fmean(classifier_acc['QDA']), 4)]})
```

[136]:
```
df_train_50 = discriminantAnalysis(50)
df_train_10k = discriminantAnalysis(10_000)

df_results = pd.concat([df_train_50, df_train_10k])
df_results.index = pd.Index(['n_train=50', 'n_train=10^4'])
df_results
```

[136]:
```
                 LDA     QDA
n_train=50    0.7276  0.7264
n_train=10^4  0.7446  0.7577
```

In the table above, we see that for `n_train = 50`, over 100 repetitions, LDA and QDA have virtually the same accuracy. I would say this result here is more or less in line with my expectations, though with as few as 50 training observations, it's not surprising that LDA performs just as well as QDA because it has lower variance. For `n_train = 10,000` though, we see that QDA barely outperforms LDA: I cannot say I am surprised at this result this time because with such a bigger training set, our model can "see" most of the variance in the data, which implies two things: - in this situation a more flexible model like QDA will likely yield better results if the clas covariance matrices are really different, because its high variance can better capture all the variability in this bigger training set while maintaining low bias, - the covariance factor present in class 2's covariance matrix will influence the data more over a much bigger training set, which means the bigger the training set, the worst LDA gets compared to QDA on average. But that difference in accuracy is still not very big in our case here because the two class covariance matrices only differ on 1 parameter out of 6.

0.1 Bayes classifier

Here we want to estimate the posterior likelihoods for both classes:

$$P(Y=1|X)P(Y=1)*P(X_1|Y=1)*P(X_2|Y=1)*P(X_3|Y=1)$$

$$P(Y=2|X)P(Y=2)*P(X_1|Y=2)*P(X_2|Y=2)*P(X_3|Y=2)$$

The actual Bayes classifier formula from Trevor Hastie's book involves the posterior probabilities, but like we wrote in the formula above it's proportional to the likelihood by $1/P(X)$ (Bayes theorem). In this case we can try to calculate the likelihood for every TEST set observation, for that we'll need approximations of the prior probabilities for classes 1 and 2, which we can take from the training set labels. Then we'll compute the likelihood for every predictor vector in the test set using the two gaussian density functions we generated our data with.

[ ]:
```
def bayesClassifier(train_set, test_set):

    mu_1, mu_2 = 3, 2
    # i had a mental bug when thinking about what the stdev should be here
    # but then I remembered that the Bayes classifier ignores the correlation␣
↪between
    # the predictors, so I just passed the diagonal elements of the covariance␣
↪matrix
    # as the stdev of the norm functions:
    sigma = 1.2
    vc = train_set['y'].value_counts()
    # prior probs:
    p_class1 = vc[1] / len(train_set['y'])
    p_class2 = vc[2] / len(train_set['y'])

    def norm1(x):
        return norm.pdf(x, loc=mu_1, scale=sigma)

    def norm2(x):
        return norm.pdf(x, loc=mu_2, scale=sigma)

    # posterior likelihoods (formula i wrote in the markdown cell above):
    L_class1 = p_class1 * norm1(test_set['x_1']) * norm1(test_set['x_2']) *␣
↪norm1(test_set['x_3'])
    L_class2 = p_class2 * norm2(test_set['x_1']) * norm2(test_set['x_2']) *␣
↪norm2(test_set['x_3'])

    # realised that there's already a built-in function (idxmax) that does the␣
↪same thing
    # def predClass(row):
    #     if row['1'] > row['2']:
    #         return 1
    #     else:
    #         return 2

    bayes_preds = pd.DataFrame({'1': L_class1, '2': L_class2}).idxmax(axis=1).
↪astype(int) # .apply(predClass, axis=1)

    return np.array(bayes_preds)
```

Now we just need to redefine our analysis function to include the Bayes classifier:

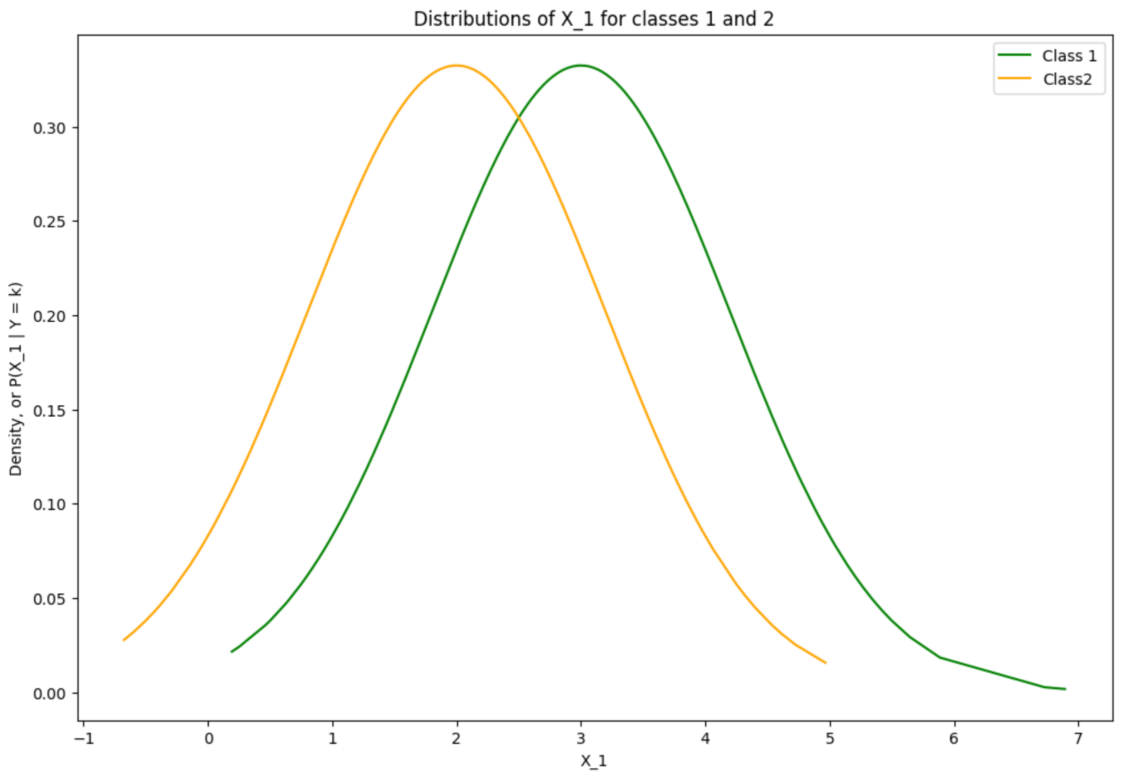
[37]:
```
def discriminantVsBayes(n_train):

    classifier_acc = {'LDA': [], 'QDA': [], 'Bayes': []}

    for _ in range(100):
        train = generateData(n_train)
        test = generateData(10_000)

        y_hat_LDA = predictLDA(train, test)
        y_hat_QDA = predictQDA(train, test)
        y_hat_Bayes = bayesClassifier(train, test)

        df_LDA = pd.DataFrame({'y': test['y'], 'y_hat': y_hat_LDA})
        df_QDA = pd.DataFrame({'y': test['y'], 'y_hat': y_hat_QDA})
        df_Bayes = pd.DataFrame({'y': test['y'], 'y_hat': y_hat_Bayes})

        classifier_acc['LDA'].append(getAccuracy(df_LDA))
        classifier_acc['QDA'].append(getAccuracy(df_QDA))
        classifier_acc['Bayes'].append(getAccuracy(df_Bayes))

    return pd.DataFrame({
        'LDA': [round(fmean(classifier_acc['LDA']), 4)]
        , 'QDA': [round(fmean(classifier_acc['QDA']), 4)]
        , 'Bayes': [round(fmean(classifier_acc['Bayes']), 4)]
    })
```

[59]:
```
df_train_50 = discriminantVsBayes(50)
df_train_10k = discriminantVsBayes(10_000)

df_results = pd.concat([df_train_50, df_train_10k])
df_results.index = pd.Index(['n_train=50', 'n_train=10^4'])
df_results
```

[59]:
```
                 LDA     QDA   Bayes
n_train=50    0.7290  0.7244  0.7425
n_train=10^4  0.7432  0.7563  0.7426
```

We see that over 100 repetitions on a large training set (10,000 obs), the accuracy of the Bayes classifier is about equal to that of LDA, and slightlty inferior to that of QDA. The reason why the Bayes classifier didn't perform better than the other methods on the n=10,000 data set might be that our two distributions actually overlap quite a lot. Let's try to quickly visualise our two gaussian density functions as a function of X_1 only:

[72]:
```
df = generateData(1000)
df.drop(columns=['x_2', 'x_3'], inplace=True)

mu_1, mu_2 = 3, 2
sigma = 1.2

def norm_(row):
    if row['y'] == 1:
        return norm.pdf(row['x_1'], loc=mu_1, scale=sigma)
    else:
        return norm.pdf(row['x_1'], loc=mu_2, scale=sigma)

df['norm'] = df.apply(norm_, axis=1)

df_class1 = df[df['y'] == 1].sort_values('x_1')
df_class2 = df[df['y'] == 2].sort_values('x_1')


plt.figure(figsize=(12, 8))
plt.plot(df_class1['x_1'], df_class1['norm'], label='Class 1', color='green')
plt.plot(df_class2['x_1'], df_class2['norm'], label='Class2', color='orange')
plt.xlabel('X_1')
plt.ylabel('Density, or P(X_1 | Y = k)')
plt.legend()
plt.title('Distributions of X_1 for classes 1 and 2')

plt.show()
```



These two curves overlap a lot! It was easy to guess just from the means being very close (2 and 3) and the diagonal elements of the covariance matrices also being identical between the two classes, but it's always nice to visualise the data we're working on. Since X_2 and X_3 follow the same distributions, our two classes indeed overlap quite a lot. That might be the main reason why the Bayes classifier wasn't able to outperform our LDA and QDA.