ECEN689-605 Computer Project 1

Shutong Jiao

March 1, 2018

Assignment 1

(a)

The error for optimal classifier in Gaussian equal-variance case is given by

$$\epsilon^* = \phi(-\frac{1}{2}\delta)$$

where ϵ is the cdf of a standard Gaussian and δ is the Mahalanobis distance between classes:

$$\delta^2 = (\mu_1 - \mu_0)^T \Sigma^{-1} (\mu_1 - \mu_0)$$

Therefore, we have:

$$\Sigma^{-1} = \frac{1}{\sigma^2(1-\rho^2)} \left(\begin{smallmatrix} 1 & -\rho \\ -\rho & 1 \end{smallmatrix}\right)$$

So, the square of Mahalanobis distance is:

$$\begin{aligned}\delta^2 &= (1,1)^T \frac{1}{\sigma^2(1-\rho^2)} \left(\begin{smallmatrix} 1 & -\rho \\ -\rho & 1 \end{smallmatrix}\right) (1,1) \\ &= \frac{2}{(1+\rho)}\sigma^2\end{aligned}$$

So the error for optimal classifier can be shown as

$$\epsilon^* = \phi(-\frac{1}{\sqrt{2}\sigma\sqrt{1+\rho}})$$

(b)

ρ is considered to be the correlation coefficient between two QSPRs X_1 and X_2 which ranges from 0 (mutually independent) to 1 (perfect correlation). As shown on Figure 1, the larger ρ, the two QSPRs will be more correlated and error rate will be higher as it is harder to distinguish two more correlated features than two less correlated features.
Meanwhile, σ reflects the variances of each QSPR. The larger σ, the two QSPRs will be more variated and thus the error rate will increase. When σ is sufficiently large like in Figure 2 when σ is close to 50, the difference of mean value in these two QSPRs will become trivial and the error rate will come close to 50%.

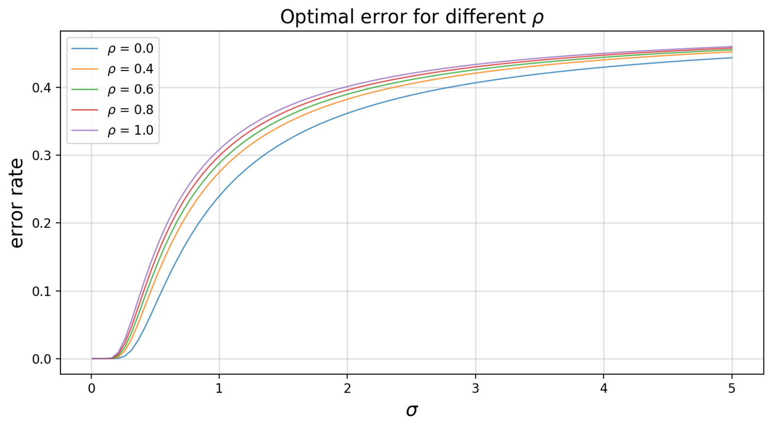

Figure 1: Optimal error when $\sigma \in [0, 5]$

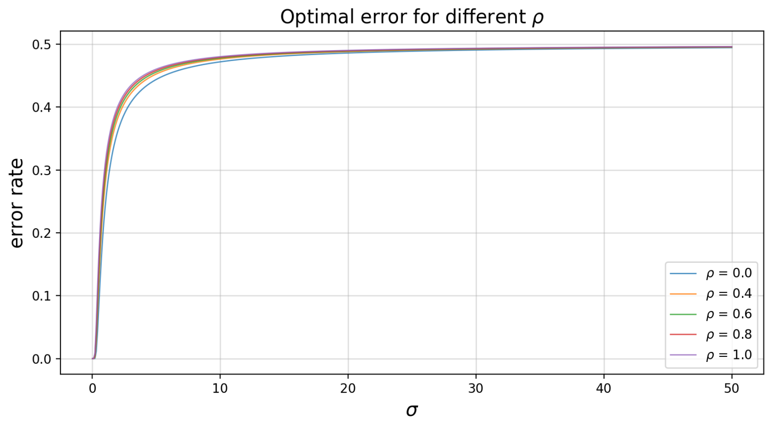

Figure 2: Optimal error when $\sigma \in [0, 50]$

(c)

We can see that LDA classifier is pretty close to the optimal classifier in Figure 3. In fact, for the given sample, I found that LDA classifier has less apparent error compared with optimal classifier. But I believe that it is just because that LDA classifier is trained on the given sample while optimal classifier can be regarded as to trained on the whole sample.

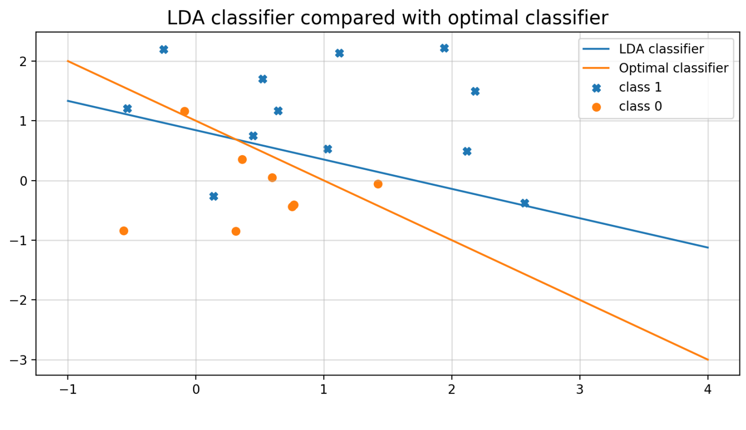

Figure 3: LDA classifier Compared with optimal classifier

(d)

Observing the classification error as a function of sample size n, firstly I found that the error given by the formula and the error form test set are very close regardless of sample size. This shows the formula can be an efficient tool for error estimation. Besides, I also observed that the error rate basically keeps unchanged when sample size increases. This means that for the problem give, 20 may be a sufficiently large sample size and simply increasing sample size can hardly do any good.

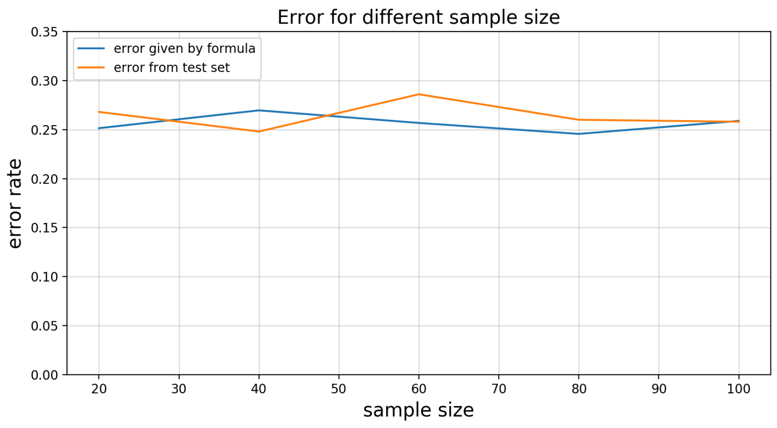

Figure 4: Error for different sample size

Python Code

(b)

```python
from scipy.stats import norm
import matplotlib.pyplot as plt
import numpy as np
import math

#define Mahanlanobis Distance
def dist(rho,sigma):
    return -1 / (sigma * math.sqrt(2) * math.sqrt(1 + rho))

# transform Mahanlanobis Distance to compute error
def disttransform(rho,sigma):
    array = np.copy(sigma)
    for item in np.nditer(array, op_flags=['readwrite']):
        item[...] = dist(rho,item)
    return array

#plot figure
plt.clf()
x = np.linspace(0.01,50,500)
plt.figure(figsize=(10, 5), dpi=200)
plt.plot(x, norm.cdf(disttransform(0,x)), lw=1, alpha=0.8, label=r'$\rho$ = '+'0.0')
plt.plot(x, norm.cdf(disttransform(0.4,x)), lw=1, alpha=0.8,label=r'$\rho$ = '+'0.4')
plt.plot(x, norm.cdf(disttransform(0.6,x)), lw=1, alpha=0.8,label=r'$\rho$ = '+'0.6')
plt.plot(x, norm.cdf(disttransform(0.8,x)), lw=1, alpha=0.8,label=r'$\rho$ = '+'0.8')
plt.plot(x, norm.cdf(disttransform(1.0,x)), lw=1, alpha=0.8,label=r'$\rho$ = '+'1.0')

plt.legend(loc='lower right')
plt.grid(linewidth=1,alpha = 0.4)
plt.title('Optimal error for different ' + r'$\rho$',fontsize=15)
plt.xlabel(r'$\sigma$',fontsize=15)
plt.ylabel('error rate',fontsize=15)
plt.savefig('1_2_50.png',dpi = 200)
plt.show()
```

(c)

```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import multivariate_normal
from scipy.stats import binom
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from numpy.linalg import inv
from scipy.stats import norm

# set parameters
n = 20
res = binom.rvs(1,0.5,size = n)
size0 = sum(res)
size1 = n - sum(res)

mean0 = np.array([0,0])
mean1 = np.array([1,1])
cov = np.array([[1,0.2],[0.2,1]])

# set samples
sample0 = multivariate_normal.rvs(mean = mean0, cov = cov, size = size0)

sample1 = multivariate_normal.rvs(mean = mean1, cov = cov, size = size1,)
```

```python
sample = np.ones((n,2))

sample[0:size1:1] = np.copy(sample1)

sample[size1:n:1] = np.copy(sample0)

sample_matrix = np.matrix(sample)

sample_mean1 = sum(sample_matrix[0:size1:1])/ size1
sample_mean1 = np.matrix(sample_mean1)

sample_mean0 = sum(sample_matrix[size1:n:1])/ size0
sample_mean0 = np.matrix(sample_mean0)

# compute covariance matrix

def covariance(matrix, mean1, mean0):
    mean1 = np.matrix(mean1)
    mean0 = np.matrix(mean0)
    len = matrix.shape[0]
    res = np.matrix(np.zeros((2,2)))
    for i in range(0,size1):
        res = res + (matrix[i] - mean1).T * (matrix[i] - mean1)
    for i in range(size1,n):
        res = res + (matrix[i] - mean0).T * (matrix[i] - mean0)
    res = res / (n - 2)
    return res

# compute other parameter
cov_matrix = covariance(sample_matrix,sample_mean1,sample_mean0)

a_n = inv(cov_matrix) * (sample_mean1 - sample_mean0).T

b_n = -0.5 * (sample_mean1 + sample_mean0) * inv(cov_matrix) * (sample_mean1 -
    sample_mean0).T

a_n_optimal = inv(np.matrix(cov)) * (np.matrix(mean1) - np.matrix(mean0)).T

b_n_optimal = -0.5 * (np.matrix(mean1) +np.matrix(mean0)) * inv(np.matrix(cov)) *
    (np.matrix(mean1) - np.matrix(mean0)).T

# define classifier and boundary function

def optimal_classifier(x):
    x = np.matrix(x)
    return a_n_optimal.T * x.T + b_n_optimal

def lda_classifier(x):
    x = np.matrix(x)
    return a_n.T * x.T + b_n

def optimal_boundary(x):
    y = (- b_n_optimal.item() -a_n_optimal[0].item() * x) / a_n_optimal[1].item()
    return y
def boundary(x):
    y = (- b_n.item() -a_n[0].item() * x) / a_n[1].item()
    return y

# plot figure

x_sample1 = np.zeros(size1)
y_sample1 = np.zeros(size1)
```

```
x_sample0 = np.zeros(size0)
y_sample0 = np.zeros(size0)
for i in range(0,size1):
    x_sample1[i] = sample1[i][0]
    y_sample1[i] = sample1[i][1]
for i in range(0,size0):
    x_sample0[i] = sample0[i][0]
    y_sample0[i] = sample0[i][1]

plt.clf()
x_range = np.linspace(-1,4,100)
y_range = np.linspace(-1,4,100)
plt.figure(figsize=(10, 5), dpi=200)

plt.scatter(x_sample1,y_sample1,marker ='X',label = 'class 1')
plt.scatter(x_sample0,y_sample0,marker ='o',label = 'class 0')
plt.plot(x_range,boundary(x_range),label = 'LDA classifier')
plt.plot(x_range,optimal_boundary(x_range),label = 'Optimal classifier')

plt.legend(loc='upper right')
plt.grid(linewidth=1,alpha = 0.4)
plt.title('LDA classifier compared with optimal classifier',fontsize=15)
plt.savefig('1_3.png',dpi = 200)
plt.show()

# estimate error in two ways

dist1 = (sample_mean0 * a_n + b_n)/(a_n.T * cov_matrix*a_n)
dist2 = - (sample_mean1 * a_n + b_n)/(a_n.T * cov_matrix*a_n)
error1 = 0.5 * ( norm.cdf(dist1) + norm.cdf(dist2))


m = 500
test = binom.rvs(1,0.5,size = m)

test_size0 = sum(test)
test_size1 = m - sum(test)

test_sample0 = multivariate_normal.rvs(mean = mean0, cov = cov, size = test_size0)
test_sample1 = multivariate_normal.rvs(mean = mean1, cov = cov, size = test_size1)


test_res_0 = np.zeros(2)
test_res_1 = np.zeros(2)
for i in range(0,test_size0):
    if lda_classifier(test_sample0[i]) > 0 :
        test_res_0[0] = test_res_0[0] + 1
    else:
        test_res_0[1]= test_res_0[1] + 1
for i in range(0,test_size1):
    if lda_classifier(test_sample1[i]) > 0 :
        test_res_1[0] = test_res_1[0] + 1
    else:
        test_res_1[1]= test_res_1[1] + 1

error2 = (test_res_0[0] + test_res_1[1]) / m

print("error 1 is ",error1.item())
print("error 2 is ",error2)
```

(d)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import multivariate_normal
from scipy.stats import binom
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from numpy.linalg import inv
from scipy.stats import norm
import math


# define function for error estimator , the function uses the same approach in question(c)

def errorestimator(n,m):
    res = binom.rvs(1,0.5,size = n)
    size0 = sum(res)
    size1 = n - sum(res)
    mean0 = np.array([0,0])
    mean1 = np.array([1,1])
    cov = np.array([[1,0.2],[0.2,1]])

    sample0 = multivariate_normal.rvs(mean = mean0, cov = cov, size = size0)

    sample1 = multivariate_normal.rvs(mean = mean1, cov = cov, size = size1,)

    sample = np.ones((n,2))

    sample[0:size1:1] = np.copy(sample1)

    sample[size1:n:1] = np.copy(sample0)

    sample_matrix = np.matrix(sample)

    sample_mean1 = sum(sample_matrix[0:size1:1])/ size1
    sample_mean1 = np.matrix(sample_mean1)

    sample_mean0 = sum(sample_matrix[size1:n:1])/ size0
    sample_mean0 = np.matrix(sample_mean0)
    def covariance(matrix, mean1, mean0):
        mean1 = np.matrix(mean1)
        mean0 = np.matrix(mean0)
        len = matrix.shape[0]
        res = np.matrix(np.zeros((2,2)))
        for i in range(0,size1):
            res = res + (matrix[i] - mean1).T * (matrix[i] - mean1)
        for i in range(size1,n):
            res = res + (matrix[i] - mean0).T * (matrix[i] - mean0)
        res = res / (n - 2)
        return res
    cov_matrix = covariance(sample_matrix,sample_mean1,sample_mean0)

    a_n = inv(cov_matrix) * (sample_mean1 - sample_mean0).T

    b_n = -0.5 * (sample_mean1 + sample_mean0) * inv(cov_matrix) * (sample_mean1 -
        sample_mean0).T

    a_n_optimal = inv(np.matrix(cov)) * (np.matrix(mean1) - np.matrix(mean0)).T

    b_n_optimal = -0.5 * (np.matrix(mean1) + np.matrix(mean0)) * inv(
        np.matrix(cov)) * (np.matrix(mean1) - np.matrix(mean0)).T
    def lda_classifier(x):
        x = np.matrix(x)
        return a_n.T * x.T + b_n
    dist1 = (mean0 * a_n + b_n)/math.sqrt(a_n.T * cov_matrix*a_n)
    dist2 = - (mean1 * a_n + b_n)/math.sqrt(a_n.T * cov_matrix*a_n)
# erro1 is the estimated error
    error1 = 0.5 * ( norm.cdf(dist1) + norm.cdf(dist2))
    m = 500
    test = binom.rvs(1,0.5,size = m)

    test_size0 = sum(test)
    test_size1 = m - sum(test)

    test_sample0 = multivariate_normal.rvs(mean = mean0, cov = cov, size = test_size0)
    test_sample1 = multivariate_normal.rvs(mean = mean1, cov = cov, size = test_size1)


    test_res_0 = np.zeros(2)
    test_res_1 = np.zeros(2)
    for i in range(0,test_size0):
        if lda_classifier(test_sample0[i]) > 0 :
            test_res_0[0] = test_res_0[0] + 1
        else:
            test_res_0[1]= test_res_0[1] + 1
    for i in range(0,test_size1):
        if lda_classifier(test_sample1[i]) > 0 :
            test_res_1[0] = test_res_1[0] + 1
        else:
            test_res_1[1]= test_res_1[1] + 1

    test_res_1
# error2 is the error on the test set
    error2 = (test_res_0[0] + test_res_1[1]) / m
    res = np.zeros(2)
    res[0] = error1.item()
    res[1] = error2
    print("error 1 is ",error1.item())
    print("error 2 is ",error2)
    return res


# plot figure

plt.clf()
m = 500
error_array = np.zeros((5,2))
n_range = np.linspace(20,100,5)
plt.figure(figsize=(10, 5), dpi=200)
plt.ylim(0,0.35)
error_array[0] = errorestimator(int(n_range[0]),m)
error_array[1] = errorestimator(int(n_range[1]),m)
error_array[2] = errorestimator(int(n_range[2]),m)
error_array[3] = errorestimator(int(n_range[3]),m)
error_array[4] = errorestimator(int(n_range[4]),m)
plt.plot(n_range,error_array[:,0],label = 'error given by formula')
plt.plot(n_range,error_array[:,1],label = 'error from test set')
plt.legend(loc='upper left')
plt.grid(linewidth=1,alpha = 0.4)
plt.xlabel('sample size',fontsize=15)
plt.ylabel('error rate',fontsize=15)
plt.title('Error for different sample size',fontsize=15)
plt.savefig('1_4_new.png',dpi = 200)
plt.show()
```

Assignment2

(a)

The aim of this preprocessing step is to extract meaningful data from the primary data for future classification and other further analysis.

1. Discard all predictors that do not have at least 60% nonzero values. The reason for this step is that predictors have too much zero values is not suitable to become predictors.
2. From the data that remain, remove the rows (samples) that contain any zero values. This is also for the same reason that samples which contain any zero values may result from technical error and thus not suitable for future classification. Also, zero value will bring extra difficulties to classifier design.
3. Randomly split the remaining data into training (20%) and test data (80%). Reject any training sample that contains more than 55% from one of the populations. The rejection process is needed because samples in which one population is significantly larger than another could be a biased sample, which means that those samples are not representative enough and will bring huge error to feature selection and classification process.

(b)

In this process, sub-optimal feature selection method is applied. We select bast individual 2 features (predictors) based on the T score. Here Welch's two sample t-tests are used because we assume that the variances are not equal, which is a quite reasonable assumption for this problem.

	T-statistics	P value
Ni	4.655815	0.000093
Fe	4.285452	0.000305
Cr	1.006328	0.324261
Mn	0.462258	0.647930
C	0.316431	0.753928

Table 1: Welch's two-sample t-test on the training data

(c)

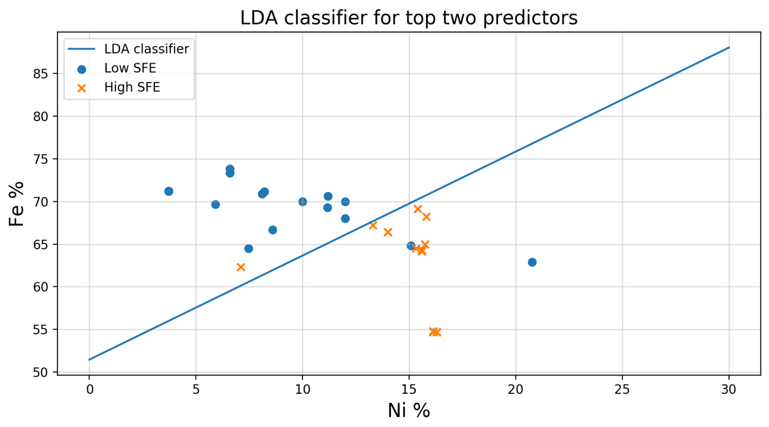

Figure 5: LDA classifier based on top two classifier

The classification error for this LDA classifier in the test set is 14.4%.

(d)

As Table2 indicates, classification error basically remains unchanged when more predictors are introduced. We could conclude that for this problem the content of Fe and Ni are the primary predictors and other predictors contributes little or even in some cases cause the error rate to increase.

Top n predictors	Classification Error
2	14.4%
3	22.7%
4	19.7%
5	19.7%

Table 2: Classification error for different predictors

Python code

```python
import matplotlib.pyplot as plt
import pandas as dp
import numpy as np
from sklearn.model_selection import train_test_split
from scipy.stats import ttest_ind
import math
from numpy.linalg import inv

# Preprocessing

SFE_res = data = dp.read_excel("SFE_Dataset.xlsx")

mask1 = SFE_res['SFE'] <= 35
mask2 = SFE_res['SFE'] >= 45
SFE_res1 = SFE_res[mask1]
SFE_res2 = SFE_res[mask2]

SFE_res_new = SFE_res1.append(SFE_res2, ignore_index=True)

header = list(SFE_res_new.columns.values)

# discard all predictors that do not have at least 60% nonzero values
n_col = SFE_res_new.shape[1]
n_row = SFE_res_new.shape[0]
nonzero_header = SFE_res_new.astype(bool).sum(axis=0);
for i in range(0,n_col):
    if nonzero_header.get(i) < 0.6 * n_row :
        del SFE_res_new[header[i]]

n_col_new = SFE_res_new.shape[1]

#  remove the rows (samples) that contain any zero values
nonzero_index = SFE_res_new.astype(bool).sum(axis=1)
nonzero_index
todrop = []
for i in range(0,n_row):
    if nonzero_index.get(i) < n_col_new :
        todrop.append(i)

drop_data = SFE_res_new.drop(SFE_res_new.index[todrop])

final_data = drop_data.reset_index(drop=True)
final_data.sort_values(['Fe'], ascending=False);

# define a split in which no population exceeds 55% in the train set

def effective_split(data):
    sample_size = data.shape[0]
    header_size = data.shape[1]
    train, test = train_test_split(data, test_size=0.8)
    def isLegal (train):
        size = train.shape[0]
        mask1 = train['SFE'] <= 35
        mask2 = train['SFE'] >= 45
        low = train[mask1]
        high = train[mask2]
        if (low.shape[0] > 0.55 * size or high.shape[0] > 0.55 * size):
            return -1
        else:
            return 1
    while isLegal(train) == -1:
        train, test = train_test_split(data, test_size=0.8)
    res = [train,test]
    return res

result = effective_split(final_data)
print(result[0].shape)
print(result[1].shape)
result[0].to_csv('train_set.csv',index = True, header = True)
result[1].to_csv('test_set.csv',index = True, header = True)

train_data = result[0]
test_data = result[1]
train_data = train_data.reset_index(drop=True)
mask_low_new = train_data['SFE'] <= 35
mask_high_new = train_data['SFE'] >= 45

# calculate T statistics and sort

feature_T = [0] * (len(train_data.columns) - 1)
feature_P = [0] * (len(train_data.columns) - 1)
feature_table = dp.DataFrame(data = list(zip(feature_T,feature_P)),
    columns=['T-statistics', 'P value'])
feature_table.index = list(train_data.columns)[0:len(list(train_data.columns)) - 1:1]
for i in feature_table.index:
    feature_table.loc[i] =
        list(ttest_ind(train_data[i][mask_low_new],train_data[i][mask_high_new],equal_var=False))

feature_table['T-statistics'] = abs(feature_table['T-statistics'])

feature_sorted = feature_table.sort_values(['T-statistics'], ascending=False)
feature_sorted

# Here we select the top two precitors

train_data_with_two = train_data[[feature_sorted.index[0],feature_sorted.index[1],'SFE']]
mask_low = train_data_with_two['SFE'] <= 35
mask_high = train_data_with_two['SFE'] >= 45
train_data_with_two_low = train_data_with_two[mask_low]
train_data_with_two_high = train_data_with_two[mask_high]
train_data_with_two_high

train_data_with_two_low =
    train_data_with_two_low[[feature_sorted.index[0],feature_sorted.index[1]]]
train_data_with_two_high =
    train_data_with_two_high[[feature_sorted.index[0],feature_sorted.index[1]]]

np_train_data_with_two_low = np.array(train_data_with_two_low.values)
np_train_data_with_two_high = np.array(train_data_with_two_high.values)
np_train_data_with_two_high

np_train_data_with_two_low_mean =
    np.matrix(sum(np_train_data_with_two_low)/np_train_data_with_two_low.shape[0])
np_train_data_with_two_high_mean =
    np.matrix(sum(np_train_data_with_two_high)/np_train_data_with_two_high.shape[0])
np_train_data_with_two_high_mean

# define covariance matrix

def covariance(low,high,mean_low,mean_high):
    res = np.matrix(np.zeros((2,2)))
    for i in range(0,low.shape[0]):
        res = res + (low[i] - mean_low).T * (low[i] - mean_low)
    for i in range(0,high.shape[0]):
        res = res + (high[i] - mean_high).T * (high[i] - mean_high)
    res = res / (low.shape[0] + high.shape[0] - 2)
    return res

# Compute LDA classifier

cov_with_two =
    covariance(np_train_data_with_two_low,np_train_data_with_two_high,np_train_data_with_two_low_mean,np_trai

a_n_two = inv(cov_with_two) * (np_train_data_with_two_high_mean -
    np_train_data_with_two_low_mean).T

b_n_two = -0.5 * (np_train_data_with_two_high_mean + np_train_data_with_two_low_mean) *
    inv(
    cov_with_two) * (np_train_data_with_two_high_mean - np_train_data_with_two_low_mean).T

def lazy_lda_classifier(a_n, b_n):
    def true_classifier(x):
        x = np.matrix(x)
        return a_n.T * x.T + b_n
    return true_classifier

def lda_classifier(a_n,b_n,x):
    x = np.matrix(x)
    return a_n.T * x.T + b_n
def boundary(a_n,b_n,x):
    y = (- b_n.item() -a_n[0].item() * x) / a_n[1].item()
    return y

# plot figure

plt.clf()
x_range = np.linspace(50,80,500)
plt.figure(figsize=(10, 5))
# plot low
plt.scatter(np_train_data_with_two_low[:,0],np_train_data_with_two_low[:,1],marker
    ='o',label = 'Low SFE')
# plot high
plt.scatter(np_train_data_with_two_high[:,0],np_train_data_with_two_high[:,1],marker
    ='x',label = 'High SFE')
plt.plot(x_range,boundary(a_n_two,b_n_two,x_range),label = 'LDA classifier')
plt.legend(loc='upper left')
plt.grid(linewidth=1,alpha = 0.4)
plt.xlabel(feature_sorted.index[0],fontsize=15)
plt.ylabel(feature_sorted.index[1],fontsize=15)
plt.title('LDA classifier for top two predictors',fontsize=15)
plt.savefig('2_1.png',dpi = 200)
plt.show()

# estimate error

test_data = result[1]
test_data_with_two = test_data[[feature_sorted.index[0],feature_sorted.index[1],'SFE']]
mask_low = test_data_with_two['SFE'] <= 35
mask_high = test_data_with_two['SFE'] >= 45
test_data_with_two_low =
    test_data_with_two[mask_low][[feature_sorted.index[0],feature_sorted.index[1]]]
test_data_with_two_high =
    test_data_with_two[mask_high][[feature_sorted.index[0],feature_sorted.index[1]]]
np_test_data_with_two_low = np.array(test_data_with_two_low.values)
np_test_data_with_two_high = np.array(test_data_with_two_high.values)

def testClassifer(a_n,b_n,test_low,test_high):
   test_res_low = np.zeros(2)
   test_res_high = np.zeros(2)
   classifier = lazy_lda_classifier(a_n,b_n)

   for i in range(0,test_low.shape[0]):
       if classifier(test_low[i]) > 0 :
           test_res_low[0] = test_res_low[0] + 1
       else:
           test_res_low[1]= test_res_low[1] + 1
   for i in range(0,test_high.shape[0]):
       if classifier(test_high[i]) > 0 :
           test_res_high[0] = test_res_high[0] + 1
       else:
           test_res_high[1]= test_res_high[1] + 1
   res_list = [test_res_low,test_res_high]
   return res_list

test_res =
    testClassifer(a_n_two,b_n_two,np_test_data_with_two_low,np_test_data_with_two_high)
error1 = (test_res[0][1] + test_res[1][0]) / (np_test_data_with_two_low.shape[0] +
    np_test_data_with_two_high.shape[0])
error = min(error1,1 - error1)

# calculate error rate for multiple preditors by function 'estimate_error'

predictors = feature_sorted.index

predictors = list(predictors)

predictors.append('SFE')
predictors

def estimate_error(num_features,train_data,test_data):
    train_data_selected = train_data[predictors[0:num_features:1]]
    train_data_selected = train_data_selected.assign( SFE = dp.Series(train_data['SFE']))

    test_data_selected = test_data[predictors[0:num_features:1]]
    test_data_selected = test_data_selected.assign( SFE = dp.Series(test_data['SFE']))

    mask_train_low = train_data_selected['SFE'] <= 35
    mask_train_high = train_data_selected['SFE'] >= 45

    mask_test_low = test_data_selected['SFE'] <= 35
    mask_test_high = test_data_selected['SFE'] >= 45

    train_data_selected_low =
        train_data_selected[mask_train_low][predictors[0:num_features:1]]
    train_data_selected_high =
        train_data_selected[mask_train_high][predictors[0:num_features:1]]

    np_train_data_selected_low = np.array(train_data_selected_low.values)
    np_train_data_selected_high = np.array(train_data_selected_high.values)

    np_train_data_selected_low_mean = np.matrix(
       sum(np_train_data_selected_low)/np_train_data_selected_low.shape[0])
    np_train_data_selected_high_mean = np.matrix(
       sum(np_train_data_selected_high)/np_train_data_selected_high.shape[0])

    test_data_selected_low =
        test_data_selected[mask_test_low][predictors[0:num_features:1]]
    test_data_selected_high =
        test_data_selected[mask_test_high][predictors[0:num_features:1]]
    np_test_data_selected_low = np.array(test_data_selected_low.values)
    np_test_data_selected_high = np.array(test_data_selected_high.values)

    def covariance(low,high,mean_low,mean_high):
        res = np.matrix(np.zeros((num_features,num_features)))
        for i in range(0,low.shape[0]):
            res = res + (low[i] - mean_low).T * (low[i] - mean_low)
        for i in range(0,high.shape[0]):
            res = res + (high[i] - mean_high).T * (high[i] - mean_high)
        res = res / (low.shape[0] + high.shape[0] - 2)
        return res

    cov_matrix =covariance(
        np_train_data_selected_low,np_train_data_selected_high,np_train_data_selected_low_mean,np_train_data_s

    a_n = inv(cov_matrix) * (np_train_data_selected_high_mean -
         np_train_data_selected_low_mean).T
    b_n = -0.5 * (np_train_data_selected_high_mean + np_train_data_selected_low_mean) *
         inv(
    cov_matrix) * (np_train_data_selected_high_mean - np_train_data_selected_low_mean).T

    test_res = testClassifer(
    a_n,b_n,np_test_data_selected_low,np_test_data_selected_high)
    error = (test_res[0][1] + test_res[1][0]) / (
        np_test_data_selected_low.shape[0] + np_test_data_selected_high.shape[0])
    error = min(error,1-error)
    return error

# show estimate error

estimate_error(2,train_data,test_data)

estimate_error(3,train_data,test_data)

estimate_error(4,train_data,test_data)

estimate_error(5,train_data,test_data)
```