Técnicas Clássicas de Reconhecimento de Padrões (2020/01)

Exercício 05 - Mistura de Gaussianas e KDE

Aluno: Ramon Gomes Durães de Oliveira (2019720188)

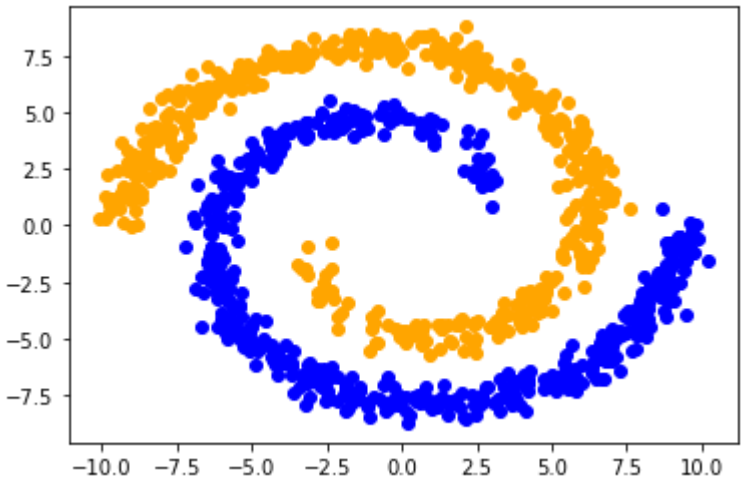
Neste exercício será implementado um classificador Bayesiano baseado em mistura de gaussianas com clustering e KDE. Suas características e propriedades serão explorados em dados sintéticos não-linearmente separáveis e, em seguida, nos dados de leucemia (Golub et al 1999).

Gerando os Dados Sintéticos: Classificação de Espirais

Abaixo, serão geradas duas classes sintéticas utilizando gaussianas multivariadas. Uma delas apresentará correlação entre os eixos.

In [31]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.cluster import KMeans
import random
import warnings
warnings.filterwarnings('ignore')
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

In [38]:
```
# Função para geração da base de dados sintética
def twospirals(n_points, n_turns, ts=np.pi, tinc=1, noise=.3):
    """
     Returns the two spirals dataset.
     modificado de:
         https://glowingpython.blogspot.com/2017/04/solving-two-spirals-problem-with-keras.html
     Primeiro gera uma espiral e obtem a segunda espelhando a primeira
    """
    # equação da espiral (coord polares): r = tinc*theta
    # n_points: número de pontos de cada espiral
    # n_turns: número de voltas das espirais
    # ts: ângulo inicial da espiral em radianos
    # tinc: taxa de crescimento do raio em função do ângulo
    # noise: desvio-padrão do ruído

    # Sorteando aleatoriamente pontos da espiral
    n = np.sqrt(np.random.rand(n_points,1))  #intervalo [0,1] equivale a [0,theta_max]
                                             #tomar a raiz quadrada ajuda a
                                             #distribuir melhor os pontos
    ns = (ts)/(2*np.pi*n_turns) #ponto do intervalo equivalente a ts radianos
    n = ns + n_turns*n # intervalo [ns,ns+n_turns] equivalente a [ts, theta_max]
    n = n*(2*np.pi) #intervalo [ts, theta_max]

    # Espiral 1
    d1x = np.cos(n)*tinc*n + np.random.randn(n_points,1) * noise
    d1y = np.sin(n)*tinc*n + np.random.randn(n_points,1) * noise

    # Espiral 2
    d2x = -np.cos(n)*tinc*n + np.random.randn(n_points,1) * noise
    d2y = -np.sin(n)*tinc*n + np.random.randn(n_points,1) * noise

    spirals_points = np.vstack((np.hstack((d1x,d1y)),np.hstack((d2x,d2y))))
    points_labels = np.hstack((np.ones(n_points),np.zeros(n_points)))
    return (spirals_points, points_labels)

X, y = twospirals(500,1,ts=np.pi,tinc=1,noise=0.4)
X1 = X[y==1,:]
X2 = X[y==0,:]
plt.plot(X1[:,0],X1[:,1],'o', c="orange")
plt.plot(X2[:,0],X2[:,1],'ob')
plt.show()
```



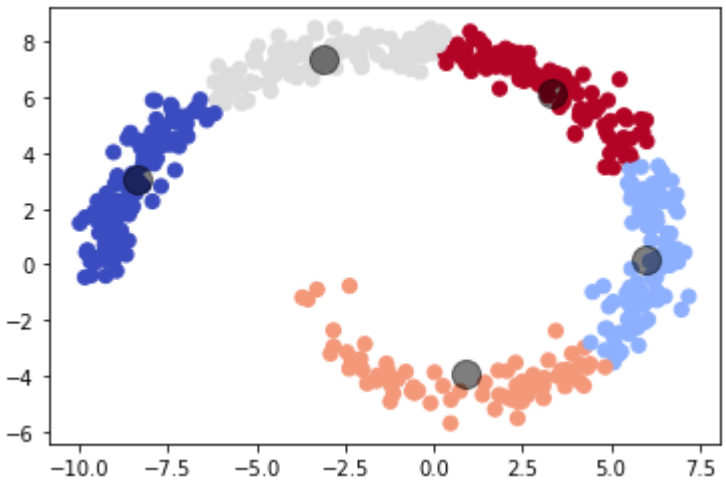
Desenvolvimento dos modelos

Classificação por Mistura de Gaussianas

Clusterização por KMeans

O primeiro passo para a implementação da mistura de Gaussianas é clusterização dos dados de entrada utilizando o KMeans. Abaixo, será implementada a clusterização para os dados sintéticos.

In [33]:
```
n_clusters = 5
kmeans = KMeans(n_clusters=n_clusters)
kmeans.fit(X1)
y_kmeans = kmeans.predict(X1)
centers = kmeans.cluster_centers_

fig = plt.figure()
ax = fig.add_subplot(111)
plt.scatter(X1[:,0], X1[:,1],  c=y_kmeans, s=50, cmap='coolwarm')
ax.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5, label='d
awa');
```



A separação por clusters foi feita com sucesso. Este código agora é a base para a classe GM (Gaussian Mixture) mostrada abaixo. Seus métodos têm a seguinte função:

- __init__: inicializa os parâmetros do modelo como a dimensão dos dados de entrada e o número de clusters. Além disso, a clusterização por KMeans é rodada nesta função, retornando também as proporções de pontos por cluster (pi), as médias e matrizes de covariância de cada cluster;
- multivariateGaussian: é o valor da gaussiana multivariada de média mu e covariância sigma avaliada em x;
- e_step : o passo E (Expectation) do algoritmo de Expectation Maximization. Este passo calcula, para cada ponto em X e cada cluster, a probabilidade relativa de que cada ponto pertença a cada cluster;
- m_step : o passo M (Maximization) do algoritmo de Expectation Maximization. Este passo otimiza os parâmetros de média e matriz de covariância para cada cluster de acordo com os valores de probabilidade relativa calculados pelo passo E.
- loglikelihood : calcula a logVerossimilhança do modelo. Este valor é utilizado apenas para garantir que o algoritmo está funcionando pois este é um valor que deve aumentar a cada iteração;
- plot : plota os pontos e as médias dos clusters resultantes;
- fit : ajsuta o modelo repetindo os passos E e M iterativamente (até o limite de iterações especificado);
- predict : estima a probabilidade para um novo ponto. Esta função é utilizada para classificar novos pontos.

In [34]:
```
from IPython import display
import time

class GM:
    def __init__(self, X, n_clusters, assume_independence=False):
        self.X = X.astype(float)
        self.n, self.d = X.shape
        self.n_clusters = n_clusters
        self.assume_independence = assume_independence

        self.r = np.zeros((self.n, self.n_clusters))

        # Clusteriza as observações
        self.kmeans = KMeans(n_clusters=self.n_clusters)
        self.kmeans.fit(self.X)
        self.y_kmeans = self.kmeans.predict(self.X)
        self.centers = self.kmeans.cluster_centers_

        # Extrai parâmetros de cada cluster
        self.mu_list = list()
        self.sigma_list = list()
        self.pi_list = list()
        for cluster in np.arange(self.n_clusters):
            Xcluster = self.X[self.y_kmeans==cluster]
            self.pi_list.append(len(Xcluster) / len(X))
            self.mu_list.append(np.mean(Xcluster, axis=0).reshape((self.d, 1
)))
            self.sigma_list.append(np.cov(Xcluster.T))

    def multivariateGaussian(self, x, mu, sigma):
        """ Função de probabilidade Gaussiana Multivariada."""
        d = max(x.shape)
        x = x.reshape((d, 1))
        mu = mu.reshape((d, 1))
        if self.assume_independence:
            sigma = np.diag(np.diag(sigma))
        term1 = 1/np.power(2*np.pi, (d / 2))
        term2 = np.power(np.linalg.det(sigma), (-1/2))
        term3 = np.exp((-1/2) * np.matmul(np.matmul((x - mu).T, np.linalg.pinv
(sigma)), (x-mu)))
        return term1 * term2 * term3

    def e_step(self):
        for i, xi in enumerate(self.X):
            xi = xi.reshape((self.d, 1))
            for c in np.arange(self.n_clusters):
                self.r[i,c] = self.pi_list[c] * self.multivariateGaussian(xi,
                                                                          self.mu_l
ist[c],
                                                                          self.sigm
a_list[c])
            self.r[i, :] = self.r[i, :] / np.sum(self.r[i, :])

    def m_step(self):
        for c in np.arange(self.n_clusters):
            mc = np.sum(self.r[:,c])
```

```
            self.pi_list[c] = mc / self.n
            self.mu_list[c] = (1/mc) * np.sum(self.X * self.r[:,c].reshape((se
lf.n, 1)), axis=0)
            # Iterando em X para calcular a matriz de covariância Sigma
            sigmac = 0
            for i, xi in enumerate(self.X):
                xi = xi.reshape((1, self.d))
                xinorm = xi - self.mu_list[c]
                sigmac += self.r[i,c] * np.dot(xinorm.T, xinorm.conj())
            self.sigma_list[c] = (1/mc) * sigmac

    def loglikelihood(self):
        loglikelihood = 0
        for i, xi in enumerate(self.X):
            xi = xi.reshape((1, self.d))
            likelihood = 0
            for c in np.arange(self.n_clusters):
                likelihood += self.pi_list[c] * self.multivariateGaussian(xi,
self.mu_list[c], self.sigma_list[c])
            loglikelihood += np.log(likelihood)
        return loglikelihood

    def plot(self, iterative_plot=False):
        if iterative_plot:
            display.clear_output(wait=True)
            fig = plt.figure()
            ax = fig.add_subplot(111)
            ax.scatter(self.X[:,0], self.X[:,1])
            ax.scatter(np.array(self.mu_list)[:,0], np.array(self.mu_list)[:,1
], c='black', s=200, alpha=0.5, label='dawa');
            display.display(plt.gcf())
            time.sleep(.1)
        else:
            plt.scatter(self.X[:,0], self.X[:,1])
            plt.scatter(np.array(self.mu_list)[:,0], np.array(self.mu_list)[:,
1], c='black', s=200, alpha=0.5, label='dawa');

    def fit(self, n_iterations = 100, iterative_plot=False):
        loglikelihood = []
        for it in np.arange(n_iterations):
            self.e_step()
            self.m_step()
            # print (self.loglikelihood())
            # Plota as iterações apenas para os casos bidimensionais
            if (self.d == 2 and iterative_plot):
                self.plot(iterative_plot)
            loglikelihood.append(np.ravel(self.loglikelihood()))
        return loglikelihood


    def predict(self, Xnew):
        d = Xnew.shape[1]
        yhat = np.zeros((Xnew.shape[0], 1))
        r_pred = np.zeros((Xnew.shape[0], self.n_clusters))
        for i, xi in enumerate(Xnew):
            xi = xi.reshape((d, 1))
            for c in np.arange(self.n_clusters):
```

```
                yhat[i] += np.ravel(self.pi_list[c] * self.multivariateGaussia
n(xi, self.mu_list[c], self.sigma_list[c]))
        return yhat
```

Agora criaremos duas instâncias da classe acima, uma para cada classe, e as treinaremos com os dados correspondentes.

In [35]:
```
# Classe 1
n_clusters = 5
GMmodel = GM(X1, n_clusters)
ll = GMmodel.fit(n_iterations = 10)

# Classe 2
n_clusters = 5
GMmodel2 = GM(X2, n_clusters)
ll2 = GMmodel2.fit(n_iterations = 10)
```

Feito o treinamento, será utilizado o método predict() para predizer as probabilidades para os pontos. A partir dessas probabilidades, teremos a classe predita para cada ponto. Abaixo é mostrada a matriz de confusão obtida:

In [36]:
```
yhat1 = GMmodel.predict(X)
yhat2 = GMmodel2.predict(X)
yhat = 1 * np.ravel(yhat1/yhat2 > (len(X2) / len(X1)))
confusion_matrix(y.astype(int), yhat)
```

Out[36]:
```
array([[499,   1],
       [  0, 500]], dtype=int64)
```

E a acurácia do modelo:

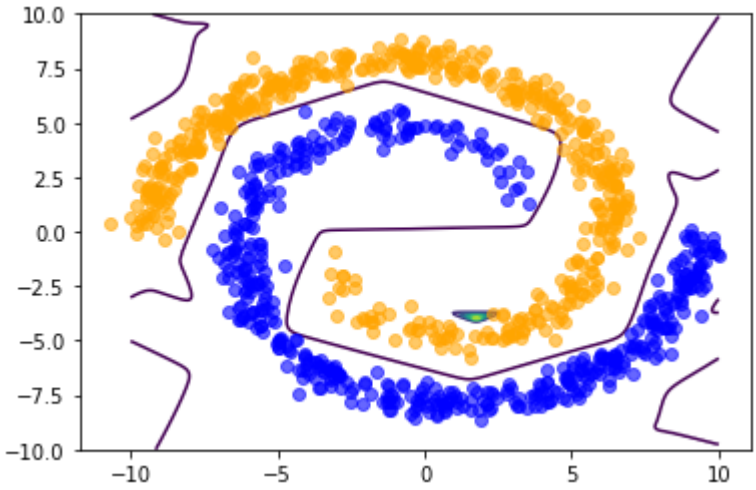
In [37]:
```
acuracia = np.sum(y.astype(int) == yhat)
acuracia / len(yhat)
```

Out[37]:
```
0.999
```

A matriz de confusão e a acurácia do classificador foram satisfatórias para o conjunto de dados sintéticos. Aqui não me preocupei em separar dados de treinamento e teste pois o intuito é apenas avaliar o funcionamento do algoritmo desenvolvido acima. Abaixo esses procedimentos serão devidamente realizados ao classificar a base de dados de leucemia (Golub et al).

Para o caso dos dados sintéticos, em duas dimensões e perfeitamente separáveis, pode-se visualizar o contorno da superfície de separação:

In [9]:
```
n_points = 240
Xgrid = np.linspace(-10, 10, n_points)
(Xg,Yg) = np.meshgrid(Xgrid, Xgrid)
Z=np.zeros( (n_points, n_points), dtype=np.float64 )
for i in range(0, n_points):
    for j in range(0, n_points):
        xin = np.array([Xg[i,j],Yg[i,j]]).reshape((1,2))
        yhat1 = GMmodel.predict(xin)
        yhat2 = GMmodel2.predict(xin)
        yhat = np.ravel((yhat1/yhat2) - (len(X2) / len(X1)))
        Z[i,j] = yhat

fig = plt.figure()
plt.contour(Xg, Yg, Z)
plt.plot(X1[:,0],X1[:,1],'o', c="orange", alpha=0.6)
plt.plot(X2[:,0],X2[:,1],'o', c="blue", alpha=0.6)
plt.show()
```



Classificação por KDE-Bayes

Abaixo será implementada uma classe que realiza o KDE para um conjunto de dados:

- a função **init** inicializa os parâmetros internos da classe e utiliza os dados de treinamento X para calcular a matriz H de acordo com a regra de Silverman
- a função silvermanH calcula a matriz H parametros dos dados de treinamento
- a função KH é a função de kernel gaussiana
- a função predict realiza a estimativa do valor de kernel para um novo ponto, levando em consideração a distribuição de treinamento.

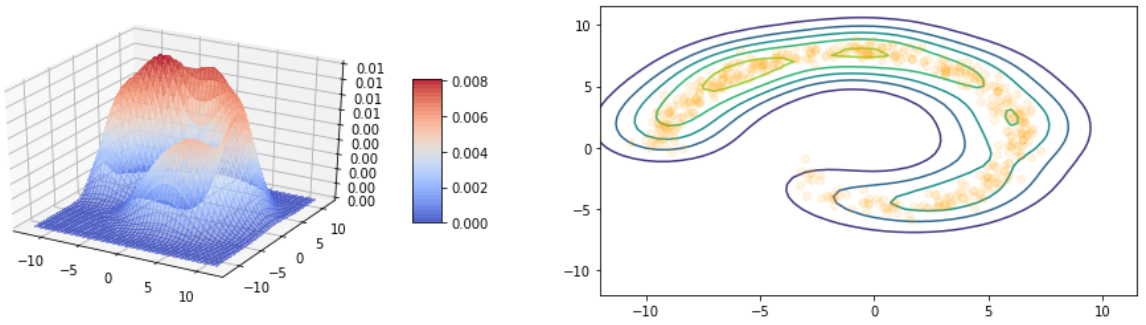
In [10]:
```
class KDE:

    def __init__(self, X):
        self.X = X
        self.n, self.d = self.X.shape
        self.sd = np.std(self.X, axis=0)
        self.H = self.silvermanH()

    def silvermanH(self):
        """Escolha de H pela regra de Silverman de acordo com os dados de entr
ada."""
        H = np.power((4 / (self.d+2)), (1/(self.d+4))) * np.power(self.n, (-1/
(self.d+4))) * self.sd
        H = np.power(H, 2)
        H = np.diag(H)
        return H

    def KH(self, x):
        """ The kernel function """
        const = np.power(2 * np.pi, -self.d/2)
        exp_term = (-1/2) * np.matmul(np.matmul(x.T, np.linalg.pinv(self.H)),
x)
        Khx = const * np.power(np.linalg.det(self.H), -1/2) * np.exp(exp_term)
        return np.ravel(Khx)

    def predict(self, Xnew):
        assert (Xnew.shape[1] == self.d)
        Yhat = np.zeros((Xnew.shape[0], 1))
        for i, xn in enumerate(Xnew):
            xn = xn.reshape(self.d, 1)
            KHi = 0
            for j, x in enumerate(self.X):
                x = x.reshape((self.d,1))
                KHi += self.KH(xn-x)
            Yhat[i] = KHi/self.n
        return Yhat
```

Testando a implementação nos dados gerados acima através do plot da superfície e do contorno de uma das classes:

In [11]:
```
Xgrid = np.arange(-12, 12, 0.5)
Ygrid = np.arange(-12, 12, 0.5)
Xgrid, Ygrid = np.meshgrid(Xgrid, Ygrid)

kde1 = KDE(X1)

Xravel = list()
for i in np.arange(Xgrid.shape[0]):
    for j in np.arange(Ygrid.shape[1]):
        Xravel.append(np.array((Xgrid[i,j], Ygrid[i,j])))
Xravel = np.array(Xravel)

Z1 = kde1.predict(Xravel)
Z1 = Z1.reshape(Xgrid.shape)
```

In [12]:
```
fig = plt.figure(figsize=plt.figaspect(0.25))
ax = fig.add_subplot(1, 2, 1, projection='3d')
surf1 = ax.plot_surface(Xgrid, Ygrid, Z1, cmap=cm.coolwarm, alpha = 0.5,
                        linewidth=0, antialiased=False)
# Customize the z axis
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf1, shrink=0.5, aspect=5)

ax = fig.add_subplot(1, 2, 2)
ax.contour(Xgrid, Ygrid, Z1)
ax.plot(X1[:,0],X1[:,1],'o', c="orange", alpha=0.1)
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x1f6c1f6f940>]
```



Classificando os pontos de acordo com o KDE:

In [13]:
```
kde2 = KDE(X2)

y1hat = kde1.predict(X)
y2hat = kde2.predict(X)
yhat = 1 * np.ravel(y1hat > y2hat)
confusion_matrix(y.astype(int), yhat)
```

Out[13]:
```
array([[490,  10],
       [ 11, 489]], dtype=int64)
```

In [14]:
```
acuracia = np.sum(y.astype(int) == yhat) / len(y)
acuracia
```

Out[14]: 0.979

Mais uma vez, a matriz de confusão e a acurácia do classificador foram satisfatórias para o conjunto de dados sintéticos. Aqui não me preocupei em separar dados de treinamento e teste ou mesmo em fazer validação cruzada do tipo leave-one-out pois o intuito é apenas avaliar o funcionamento do algoritmo desenvolvido acima. Abaixo esses procedimentos serão devidamente realizados ao classificar a base de dados de leucemia (Golub et al).

Classificação da Base de Dados de Leucemia (Golub et al)

O carregamento e seleção de variáveis dessa base de dados foi explicado nos trabalhos anteriores. Desta vez, os dados serão apenas carregados sem muita explicação por economia de espaço.

In [30]:
```
# Carregando os dados
class_df = pd.read_csv('./data/actual.csv')
train_df = pd.read_csv('./data/data_set_ALL_AML_train.csv')
test_df = pd.read_csv('./data/data_set_ALL_AML_independent.csv')

# Tratando dados de treinamento
valid_columns = [col for col in train_df.columns if "call" not in col]
train_df = train_df[valid_columns]
train_df = train_df.T
train_df = train_df.drop(['Gene Description','Gene Accession Number'],axis=0)
train_df.index = pd.to_numeric(train_df.index)
train_df.sort_index(inplace=True)
class_dict = {'AML':0,'ALL':1}
train_df['class'] = class_df[:38]['cancer'].replace(class_dict).values

# Tratando dados de teste
valid_columns = [col for col in test_df.columns if "call" not in col]
test_df = test_df[valid_columns]
test_df = test_df.T
test_df = test_df.drop(['Gene Description','Gene Accession Number'],axis=0)
test_df.index = pd.to_numeric(test_df.index)
test_df.sort_index(inplace=True)
test_df['class'] = class_df[38:]['cancer'].replace(class_dict).values

# Selecionando os 50 genes mais relevantes
mu1 = train_df[train_df['class']==0].iloc[:,:-1].mean()
sigma1 = train_df[train_df['class']==0].iloc[:,:-1].std()
mu2 = train_df[train_df['class']==1].iloc[:,:-1].mean()
sigma2 = train_df[train_df['class']==1].iloc[:,:-1].std()
Pgc = (mu1 - mu2) / (sigma1 + sigma2)
abs_Pgc = np.abs(Pgc)
selected_genes = abs_Pgc>.91
# selected_genes = abs_Pgc>1.3
selected_genes = selected_genes.index[selected_genes.values]

# Transformando para o formato de entradas e saídas X, y
Xtrain = np.array(train_df[selected_genes])
ytrain = np.array(train_df['class'])
Xtest = np.array(test_df[selected_genes])
ytest = np.array(test_df['class'])
```

Classificação por GMM - Bayes

Ajustando os modelos nos dados de treinamento:

In [16]:
```
# Classe 1
X1 = Xtrain[ytrain==0]
n_clusters = 3
GMmodel = GM(X1, n_clusters, assume_independence=True)
ll = GMmodel.fit(n_iterations = 10)

# Classe 2
X2 = Xtrain[ytrain==1]
n_clusters = 3
GMmodel2 = GM(X2, n_clusters, assume_independence=True)
ll2 = GMmodel2.fit(n_iterations = 10)
```

Realizando as predições para cada classe e calculando a matriz de confusão

In [25]:
```
yhat1 = GMmodel.predict(Xtest.astype(float))
yhat2 = GMmodel2.predict(Xtest.astype(float))
yhat = 1 * np.ravel(yhat1/yhat2 < (len(X2) / len(X1)))
confusion_matrix(ytest.astype(int), yhat)
```

Out[25]:
```
array([[13,  1],
       [ 1, 19]], dtype=int64)
```

Calculando a acurácia

In [26]:
```
acuracia = np.sum(ytest.astype(int) == yhat) / len(yhat)
acuracia
```

Out[26]:
```
0.9411764705882353
```

O resultado da classificação por mistura de Gaussianas foi satisfatório. É possível perceber pela matriz de confusão que houve apenas uma observação errada por classe, resultando numa acurácia de 94.12%.

Classificação por KDE - Bayes

In [27]:
```
kde1 = KDE(X1.astype(float))
kde2 = KDE(X2.astype(float))
```

In [28]:
```
y1hat = kde1.predict(Xtest.astype(float))
y2hat = kde2.predict(Xtest.astype(float))
yhat = 1 * np.ravel(y1hat < y2hat)
confusion_matrix(ytest.astype(int), yhat)
```

Out[28]:
```
array([[13,  1],
       [ 0, 20]], dtype=int64)
```

In [29]:
```
acuracia = np.sum(ytest.astype(int) == yhat) / len(ytest)
acuracia
```

Out[29]: 0.9705882352941176

O resultado da classificação pelo KDE - Bayes foi satisfatório e ligeiramente superior ao obtido acima pelo método da mistura de gaussianas. É possível perceber pela matriz de confusão que houve apenas uma observação errada, resultando numa acurácia de 97.06%.