Solução Lista 05

Nome: Julia Xavier
E-mail: julia.xavier@aluno.ufabc.edu.br
Nome: Leonardo Bernardes Lério
E-mail: leonardo.lerio@aluno.ufabc.edu.br

14 de novembro, 2024

Exercício 01

```
import numpy as np
import matplotlib.pyplot as plt
```

```
np.random.seed(1234)
```

```
x1 = np.random.uniform(-4, 4, size=100)
x1
```

```
## array([-2.4678444 ,  0.97687017, -0.49817809,  2.28286867,  2.23980646,
##        -1.81925916, -1.78828596,  2.41497742,  3.66511483,  3.00746108,
##        -1.13746184,  0.007961  ,  1.46770348,  1.70161622, -1.03799396,
##         0.48956949,  0.02466532, -3.8898524 ,  2.18261297,  3.06112953,
##        -1.08091213,  0.92316943, -3.39695007, -1.04940795,  3.46512082,
##         1.21102515, -0.82237938,  2.30984114, -1.46531102,  0.54478922,
##         2.95301912, -0.51061261,  2.41718114, -2.8498654 ,  1.63408777,
##         1.63665047, -2.24966315,  3.39894103, -0.46287396,  3.27452767,
##        -3.52152622, -2.52570333, -3.62115777,  1.39904755,  0.75699824,
##         0.2664813 , -3.6534075 ,  0.49146464, -1.36265244,  0.02373466,
##        -3.10484546,  0.85754965,  0.52755714, -3.9458875 ,  0.93953367,
##         3.29698309,  2.32419306,  3.93665173,  3.6704141 ,  2.33571308,
##        -1.71799232,  0.99933364, -0.17524963, -2.43459857, -0.94146038,
##        -3.56901052, -0.38681273,  3.85603793, -3.0084584 , -3.04495282,
##         1.90818445,  0.69842907, -0.22693973, -3.14298546, -2.16625148,
##         3.19972156, -0.6659717 ,  0.2868133 , -3.95033187, -1.59486635,
##        -0.50485462,  0.89719198,  3.3455846 ,  1.00589336,  1.64798052,
##        -2.80133027,  1.96850727,  2.64805594,  1.06980615, -0.49352095,
##        -2.7794178 ,  0.54727692,  0.22579422,  3.61143011, -0.15712657,
##         0.02047651,  0.29502554,  2.55361654, -3.5430749 ,  1.35537394])
```

```
y = x1**3 - 2*x1**2 + 5*x1 + 13 + np.random.normal(0, 5.0, size=100)y
```

```
## array([-2.27126943e+01,  2.18326025e+01,  1.12432875e+01,  3.28484414e+01,
```

```
##           2.58012889e+01, -1.07367356e+01, -1.31954948e+01,  2.45714925e+01,
##           5.77761424e+01,  3.67398764e+01,  1.52955161e+00,  1.56811195e+01,
##           1.38469251e+01,  1.80847050e+01,  5.99282671e+00,  1.79184985e+01,
##           1.56400837e+01, -9.41418559e+01,  2.72044370e+01,  4.50663842e+01,
##           9.02644602e-02,  1.43580388e+01, -6.01387412e+01, -2.01076314e+00,
##           5.22947785e+01,  9.34445287e+00,  4.72548029e+00,  2.99481407e+01,
##          -2.78671485e+00,  1.43811693e+01,  3.94790091e+01,  6.99861938e-01,
##           2.77587551e+01, -3.86644170e+01,  1.89512041e+01,  1.71214420e+01,
##          -2.31702143e+01,  4.83376793e+01,  1.64288994e+00,  4.50073544e+01,
##          -7.54775089e+01, -2.99938621e+01, -7.53442978e+01,  2.22121214e+01,
##           1.72704736e+01,  1.49654385e+01, -7.66446154e+01,  2.45606277e+01,
##           3.14105577e+00,  8.30741587e+00, -6.21615992e+01,  2.60988339e+01,
##           6.55123615e+00, -9.32551551e+01,  2.07487469e+01,  4.16841942e+01,
##           2.98850263e+01,  5.84442585e+01,  5.97398506e+01,  2.38883863e+01,
##          -3.05906839e+00,  2.19182754e+01,  1.14483026e+01, -1.36292137e+01,
##           8.16625620e+00, -7.17992214e+01,  8.33870719e+00,  5.95941921e+01,
##          -4.05839707e+01, -5.30244117e+01,  1.15885052e+01,  1.41897227e+01,
##           7.31701356e+00, -5.18481873e+01, -1.46980793e+01,  3.75624669e+01,
##           6.88671486e+00,  9.71214225e+00, -1.03905655e+02, -2.98830167e+00,
##           1.29811732e+01,  1.75307225e+01,  4.95513575e+01,  2.19642963e+01,
##           1.99208325e+01, -4.14378714e+01,  1.80297383e+01,  2.45892199e+01,
##           1.79828540e+01,  8.80997107e+00, -2.72004127e+01,  1.59126437e+01,
##           6.99135802e+00,  5.91890406e+01,  1.42183517e+00,  6.36388998e+00,
##           1.61445494e+01,  2.93044320e+01, -6.79377230e+01,  1.13448338e+01])
```

(a) Implemente uma função que retorna a matriz kernel K utilizando o kernel polinomial de grau 3 com constante c = 1. Ou seja, calcule a matriz K (pertence) R m×m tal que Kij = (xi1 * xj1 + 1)^3

```python
def calcula_kernel_matriz(x, degree):

    matriz = len(x)
    kernel = np.zeros((matriz, matriz))
    for i in range(matriz):
        for j in range(matriz):
            kernel[i, j] = (x[i] * x[j] + 1)**degree

    return kernel
```

(b) Calcule os coeficientes (alpha) de acordo com o método da regressão Ridge com Kernel com penalização (lambda) = 0.001. Lembre-se que são dados por (alpha) = ((K + (lambda)I)^-1)y.

```python
def calcula_alpha(kernel, y, lambda_value):
    matriz = kernel.shape[0]
    alpha = np.linalg.inv(kernel + lambda_value*np.identity(matriz)).dot(y)
    return alpha
```

```python
kernel = calcula_kernel_matriz(x1, 3)
kernel
```

```
## array([[ 3.56439431e+02, -2.80777764e+00,  1.10810059e+01, ...,
##          -1.49039554e+02,  9.25080240e+02, -1.28927721e+01],
##         [-2.80777764e+00,  7.46375279e+00,  1.35278013e-01, ...,
```

```
##          4.26750907e+01, -1.49073544e+01,  1.25522631e+01],
##         [ 1.10810059e+01,  1.35278013e-01,  1.94461274e+00, ...,
##          -2.01582487e-02,  2.11409343e+01,  3.42592191e-02],
##         ...,
##         [-1.49039554e+02,  4.26750907e+01, -2.01582487e-02, ...,
##          4.25421455e+02, -5.21204303e+02,  8.87825120e+01],
##         [ 9.25080240e+02, -1.49073544e+01,  2.11409343e+01, ...,
##          -5.21204303e+02,  2.48967591e+03, -5.49669861e+01],
##         [-1.28927721e+01,  1.25522631e+01,  3.42592191e-02, ...,
##          8.87825120e+01, -5.49669861e+01,  2.28347206e+01]])
```

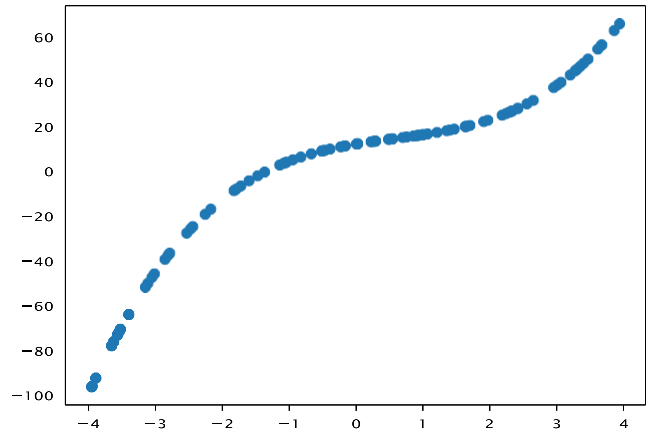
```
lambda_value = 0.001
alpha = calcula_alpha(kernel, y, lambda_value)
alpha
```

```
## array([ 3.31380704e+03, 5.93912611e+03, 2.45785447e+03, 6.92307085e+03, ##
4.11907009e+02, -1.77201027e+03, -4.88080989e+03, -3.11888912e+03, ##
1.90444161e+03, -1.43520965e+03, -9.33251167e+02, 3.84337544e+03, ## -
4.61670346e+03, -2.01350538e+03, 2.30905586e+03, 4.00362183e+03, ##
3.72085997e+03, -1.46030958e+03, 2.49836876e+03, 5.70701294e+03, ## -
3.07882550e+03, -1.30210790e+03, 4.16543014e+03, -5.55942263e+03, ##
2.57548647e+03, -7.65941419e+03, -1.28275371e+03, 3.67739658e+03, ## -
4.70460705e+02, 2.48699257e+02, 2.46229292e+03, -7.99290683e+03, ##
3.73351216e+01, 9.12083518e+02, -6.40560715e+02, -2.48900205e+03, ## -
3.66165125e+03, 4.99012236e+02, -7.40022008e+03, 5.04412117e+02, ## -
4.57802833e+03, -2.09518315e+03, 1.10465996e+03, 4.16926300e+03, ##
2.29856085e+03, 1.95662575e+03, 1.65406370e+03, 1.06382620e+04, ##
3.82879639e+03, -3.60729299e+03, -1.18877825e+04, 1.07157771e+04, ## -
7.51344416e+03, 3.05218945e+03, 5.01817421e+03, -3.40191677e+03, ##
3.42743029e+03, -6.97824820e+03, 3.69524393e+03, -2.72075607e+03, ##
3.83071759e+03, 5.92519784e+03, 5.73119975e+02, 1.13493451e+04, ##
3.38953022e+03, 1.71424783e+03, -1.23338988e+03, -2.84289353e+03, ##
5.48217972e+03, -5.38900052e+03, -1.02580489e+04, -5.48109686e+02, ## -
3.26069287e+03, 1.47103709e+02, 2.53592753e+03, -5.05635774e+03, ## -
5.59221662e+02, -3.38210570e+03, -7.30709175e+03, 1.56878785e+03, ##
4.24536978e+03, 1.98124334e+03, 3.17474179e+03, 5.94189549e+03, ##
2.27302204e+02, -3.74271383e+03, -4.38939332e+03, -6.69383431e+03, ##
1.66869895e+03, -9.90949273e+00, 9.66108909e+03, 1.77039288e+03, ## -
5.84411019e+03, 5.03974384e+03, -9.55473694e+03, -5.53499002e+03, ##
3.01600175e+03, -4.45749048e+02, 4.14107471e+03, -6.44259381e+03])
```

(c) Faça o gráfico em duas dimensões da função resultante usando x = x1 e y = "valor predito". Lembre-se que o Kernel Ridge regression faz predições da seguinte forma: f(x) = (somatoria) m, i=1 (alpha)i k (xi,x) + b

```
x = x1
```

```
preditor = np.dot(kernel, alpha)
plt.scatter(x, preditor)
plt.show()
```



Exercício 02

```
np.random.seed(1234)
x1 = np.random.uniform(-10, 10, size=100)
y = np.where(x1 == 0, 1, np.sin(x1) / x1) + np.random.normal(0, 0.05, size=100)
```

a) Implemente uma função que retorna a matriz kernel K utilizando o kernel polinomial de grau 3 com constante c = 1. Ou seja, calcule a matriz K (pertence) R m×m tal que Kij = (xi1 * xj1 + 1)^3

```
def calcula_kernel_matriz(x, sigma):
    matriz = len(x)
    kernel = np.zeros((matriz, matriz))
    for i in range(matriz):
        for j in range(matriz):
            kernel[i, j] = np.exp(-((x[i] - x[j])**2) / (2 * sigma**2))
    return kernel
```

(b) Calcule os coeficientes (alpha) de acordo com o método da regressão Ridge com Kernel com penalização (lambda) = 0.001. Lembre-se que são dados por (alpha) = ((K + (lambda)I)^-1)y. Para simplificar, faça o intercepto-y ser b = 0.

```
def calcula_alpha(kernel, y, valor_lamb):
    matriz = kernel.shape[0]
    alpha = np.linalg.inv(kernel + valor_lamb*np.identity(matriz)).dot(y)
    return alpha
```

```
sigma = 1
kernel = calcula_kernel_matriz(x1, sigma)
kernel
```

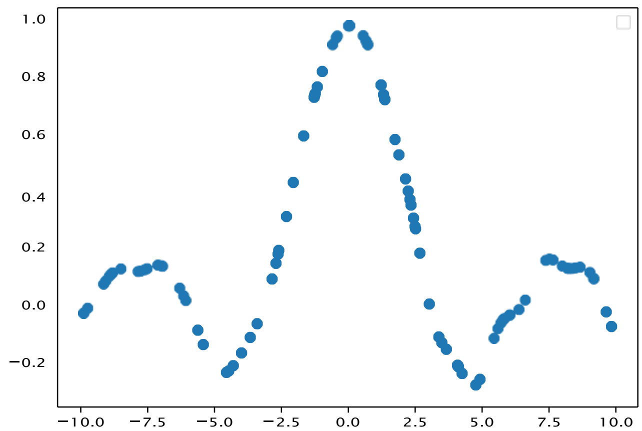
```
## array([[1.00000000e+00, 7.86571914e-17, 5.42927865e-06, ...,
##         6.00975990e-35, 2.69741319e-02, 1.45292910e-20],
##        [7.86571914e-17, 1.00000000e+00, 1.11458516e-03, ...,
```

```
##          4.22570835e-04, 1.87561114e-28, 6.39094170e-01],
##         [5.42927865e-06, 1.11458516e-03, 1.00000000e+00, ...,
##          2.29119523e-13, 2.61298158e-13, 2.17385733e-05],
##         ...,
##         [6.00975990e-35, 4.22570835e-04, 2.29119523e-13, ...,
##          1.00000000e+00, 3.58477137e-51, 1.12562799e-02],
##         [2.69741319e-02, 1.87561114e-28, 2.61298158e-13, ...,
##          3.58477137e-51, 1.00000000e+00, 2.72251158e-33],
##         [1.45292910e-20, 6.39094170e-01, 2.17385733e-05, ...,
##          1.12562799e-02, 2.72251158e-33, 1.00000000e+00]])
```

```
valor_lamb = 0.001
alpha = calcula_alpha(kernel, y, valor_lamb)
alpha
```

```
## array([ -4.89939518,  20.09526603,  41.73563898,  42.94095795,
##         -21.20068414,   2.80304563, -33.89659859, -28.431216  ,
##         -16.21400782, -29.33117349,   3.03614357,  57.43144193,
##         -30.46453991,   6.94040395,  33.63253042,  30.69586415,
##          55.02991929,   1.36801372,  10.27574776,  48.16010087,
##         -18.4503964 , -59.64305247,  38.6045235 , -44.44504639,
##           0.18034168, -44.68639574, -18.33061705,  12.95777087,
##         -26.98096475,  -7.69196524,   8.19747622, -64.25973173,
##           3.85743629,  -5.69297003,   4.41927658, -13.61291476,
##         -49.47336139,  -4.8502502 , -52.89168134,  15.59013857,
##         -61.37787541, -61.24055157,  -1.22803181,  71.35101497,
##         -11.0249174 ,  16.41558514,   7.22264538,  97.03295866,
##          26.84488117, -18.18450133, -85.46680822,  60.37309397,
##         -84.8952311 ,  51.30175097,   5.08423464, -25.35929246,
##          12.66551254,  -2.44724855,   2.7349569 , -46.63588492,
##          39.01128652,  24.93231352,  34.38487162,  77.43727246,
##          36.93295172,   1.81042866,  15.24432585,   5.41508239,
##          75.65992186, -26.76565588, -32.52433524, -29.94702423,
##          -2.40424889,  35.80308798,  29.07547896, -40.60499239,
##          -7.66815736, -38.40652129, -51.90812424,  -3.88602273,
##          58.81678851, -27.96765943,  33.29125531,  26.7456605 ,
##          15.7291672 , -61.64729783,  17.90844239, -22.52037727,
##           3.62875417,  17.60443281,  68.80423388,   7.44942847,
##         -58.2688743 ,   8.66144342, -67.59603198, -37.22800063,
##          25.04623548,  37.17317884,  25.59897238, -27.37112653])
```

```
x = x1
preditor = np.dot(kernel, alpha)
plt.clf()
plt.scatter(x, preditor)
plt.legend()
plt.show()
```



Exercício 03

(a) Se a fronteira de decisão de Bayes é linear, qual dos algoritmos LDA ou QDA você espera ter melhor performance segundo o conjunto de treinamento? E sobre o conjunto de testes? Justifique sua resposta.

R: O algoritmo de Análise Discriminante Linear (LDA) é mais adequado do que o algoritmo de Análise Discriminante Quadrática (QDA) em termos de desempenho, tanto no conjunto de treinamento quanto no conjunto de testes. Pois o LDA assume que as classes têm a mesma matriz de covariância, o que reduz a complexidade do modelo em comparação com o QDA, que permite que cada classe tenha sua própria matriz de covariância. Com uma fronteira de decisão linear, o LDA é mais provável de se ajustar bem aos dados de treinamento e testes, pois é menos suscetível a overfitting.

(b) Se a fronteira de decisão de Bayes é não-linear, qual dos algoritmos LDA ou QDA você espera ter melhor performance segundo o conjunto de treinamento? E sobre o conjunto de testes? Justifique sua resposta.

R: O algoritmo de Análise Discriminante Quadrática (QDA) tem maior probabilidade de ter melhor desempenho do que o algoritmo de Análise Discriminante Linear (LDA), tanto no conjunto de treinamento quanto no conjunto de testes. Pois o QDA permite que cada classe tenha sua própria matriz de covariância, o que lhe confere mais flexibilidade para aprender relações não-lineares nos dados.

Exercício 04

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB

url = "https://raw.githubusercontent.com/hargurjeet/MachineLearning/Ionosphere/ionosphere_data.csv"
df = pd.read_csv(url)
```

```
columns_to_remove = ["column_a", "column_b"]
df = df.drop(columns=columns_to_remove)

df.head()
```

```
##    column_c  column_d  column_e  ...  column_ag  column_ah  column_ai
## 0   0.99539  -0.05889   0.85243  ...    0.18641   -0.45300          g
## 1   1.00000  -0.18829   0.93035  ...   -0.13738   -0.02447          b
## 2   1.00000  -0.03365   1.00000  ...    0.56045   -0.38238          g
## 3   1.00000  -0.45161   1.00000  ...   -0.32382    1.00000          b
## 4   1.00000  -0.02401   0.94140  ...   -0.04608   -0.65697          g
##
## [5 rows x 33 columns]
```

```
X = df.drop(columns=["column_ai"])
y = df["column_ai"]

X
```

```
##      column_c  column_d  column_e  ...  column_af  column_ag  column_ah
## 0     0.99539  -0.05889   0.85243  ...   -0.54487    0.18641   -0.45300
## 1     1.00000  -0.18829   0.93035  ...   -0.06288   -0.13738   -0.02447
## 2     1.00000  -0.03365   1.00000  ...   -0.24180    0.56045   -0.38238
## 3     1.00000  -0.45161   1.00000  ...    1.00000   -0.32382    1.00000
## 4   1.00000  -0.02401   0.94140  ...   -0.59573   -0.04608   -0.65697##
..        ...       ...       ...  ...        ...        ...        ...##
346   0.83508   0.08298   0.73739  ...   -0.10714    0.90546   -0.04307
## 347   0.95113   0.00419   0.95183  ...   -0.00035    0.91483    0.04712
## 348   0.94701  -0.00034   0.93207  ...    0.00442    0.92697   -0.00577
## 349   0.90608  -0.01657   0.98122  ...   -0.03757    0.87403   -0.16243
## 350   0.84710   0.13533   0.73638  ...   -0.06678    0.85764   -0.06151
##
## [351 rows x 32 columns]
```

```
y
```

```
## 0      g
## 1      b
## 2      g
## 3      b
## 4      g
##       ..
## 346    g
## 347    g
## 348    g
## 349    g
## 350    g
## Name: column_ai, Length: 351, dtype: object
```

```
lda = LinearDiscriminantAnalysis()
qda = QuadraticDiscriminantAnalysis()
nb = GaussianNB()
```

```
k_fold = 10
lda_scores = cross_val_score(lda, X, y, cv=k_fold)
qda_scores = cross_val_score(qda, X, y, cv=k_fold)
nb_scores = cross_val_score(nb, X, y, cv=k_fold)
```

```
print("Média Acuracia LDA:", np.mean(lda_scores))
```

```
## Média Acuracia LDA: 0.8377777777777776
```

```
print("Média Acuracia QDA:", np.mean(qda_scores))
```

```
## Média Acuracia QDA: 0.8775396825396825
```

```
print("Média Acuracia Naive Bayes:", np.mean(nb_scores))
```

```
## Média Acuracia Naive Bayes: 0.7975396825396825
```