Exercício KNN

Introdução ao Reconhecimento de Padrões

Turma TB

Gabriel Vaz Cançado Ferreira

Matrícula: 2021015194

Parte 1

Para implementar o algoritmo do KNN, inicialmente constrói-se uma função para o cálculo de distância euclidiana entre dois vetores.

Cálculo da Distância eucliadiana

```python
def euclidean_distance(x, y):
    if len(x) != len(y):
        raise ValueError("x and y must have the same dimension")
    return np.sqrt(sum((xi - yi) ** 2 for xi, yi in zip(x, y)))
```

Em seguida, realizamos a implementação do KNN, conforme apresentado em aula.

Implementação do KNN

```python
def knn(X, Y, x_test, k):
    N = len(X)
    predictions = []

    for x in x_test:
        distances = []
        for i in range(N):
            d = euclidean_distance(x, X[i])
            distances.append((d, Y[i]))

        distances.sort(key=lambda x: x[0])
        neighbors = distances[:k]

        classes = [c for _, c in neighbors]

        pred = Counter(classes).most_common(1)[0][0]
        predictions.append(pred)

    return predictions
```

Para cada vetor da matriz x_test, calculamos a distância euclidiana entre o vetor e cada vetor da matriz de treino X. Em seguida, as distâncias são ordenadas e selecionamos os k primeiros vizinhos e contamos as classes dos vizinhos de modo a classificar os dados de teste.

Parte 2

Para executar a parte 2, executamos dois loops. O primeiro visa variar os valores de k entre {2,4,6,8,.., 20} e o segundo variar o desvio padrão das duas distribuições gaussianas utilizadas para realizar os experimentos entre {0.1,0.3,0.5,0.7,...,2.5} mantendo k = 4. Por fim, variamos ambos os parâmetros simultaneamente.

Parâmetros e Loops

```
# Parâmetros
k_values = range(2,22,2)
std_values = np.arange(0.1, 2.6, 0.2)
nc = 100
seqi = np.arange(0, 6.1, 0.1)
seqj = np.arange(0, 6.1, 0.1)
# ------------------------------
# 1) Fixar k, variar std
# ------------------------------
k_fixed = 4
fig, axes = plt.subplots(1, len(std_values), figsize=(20, 4))
for i, std in enumerate(std_values):

# ------------------------------
# 2) Fixar std, variar k
# ------------------------------
std_fixed = 0.95
fig, axes = plt.subplots(1, len(k_values), figsize=(20, 4))
for i, k in enumerate(k_values):

# ------------------------------
# 3) Variar k e std ao mesmo tempo (grade)
# ------------------------------
for j, k in enumerate(k_values):
    fig, axes = plt.subplots(1, len(std_values), figsize=(3*len(std_values), 4))
    for i, std in enumerate(std_values):
```

Em seguida, definimos parâmetros como o número de vetores utilizados e a dimensão do problema, partindo então para a criação das duas distribuições gaussianas, uma atribuída a classe 1 e outra ao vetor de classe -1, juntamente com seu acoplamentos na

matriz X e no vetor de classes Y. Em seguida monta-se um grid e retira-se amostras. As amostras são usadas como dados de teste e são classificadas pelo algoritmo.

Gaussianas e predições

```
    ax = axes[i]
    xc1 = np.random.normal(0, std_fixed, (nc, 2)) + np.array([2, 2])
    xc2 = np.random.normal(0, std_fixed, (nc, 2)) + np.array([4, 4])
    X = np.vstack((xc1, xc2))
    Y = np.array([1]*nc + [-1]*nc)

    M1 = np.zeros((len(seqi), len(seqj)))
    for ci, xi in enumerate(seqi):
        for cj, yj in enumerate(seqj):
            xt = np.array([xi, yj])
            M1[ci, cj] = knn(X, Y, xt, k)
```

Por fim, plotamos os resultados.

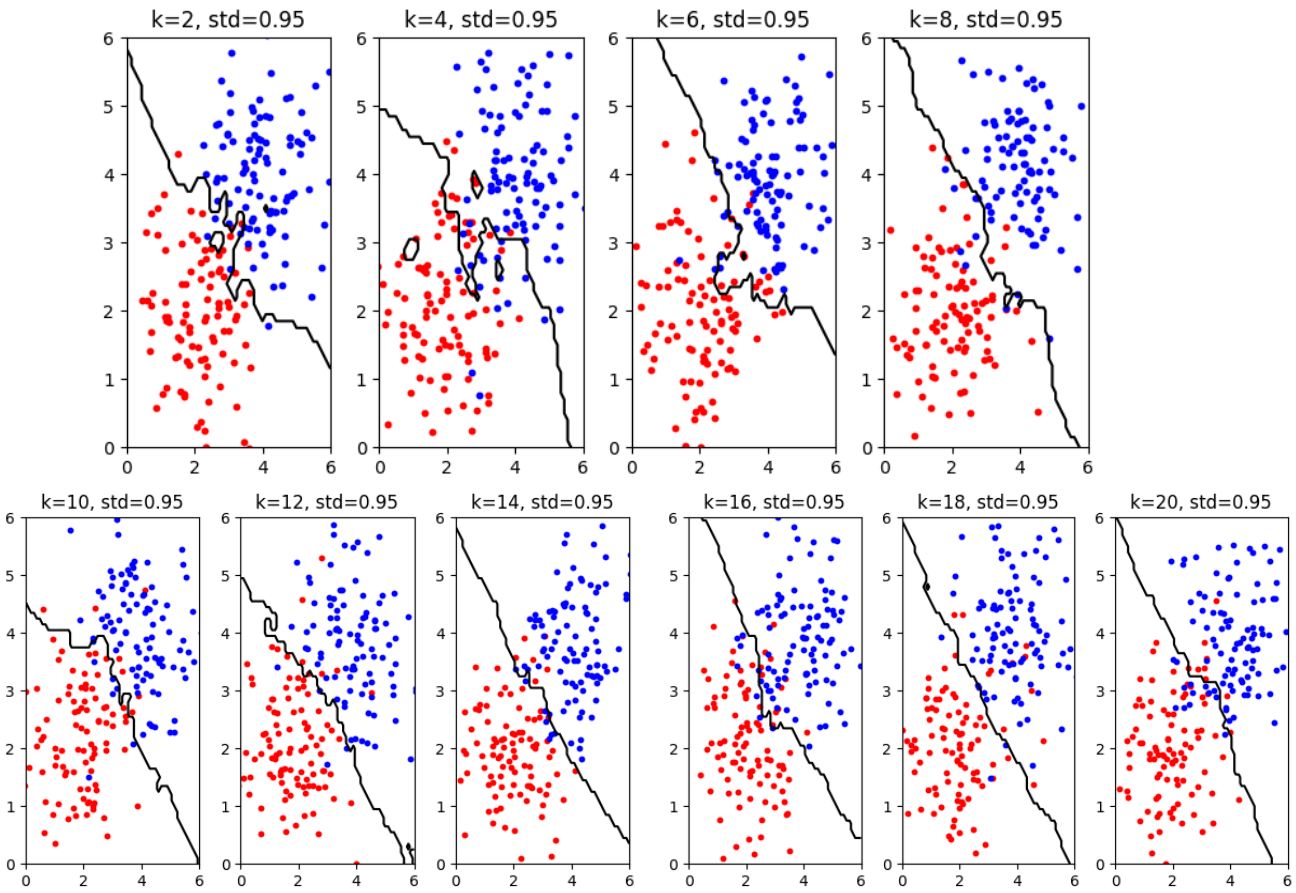
Gaussianas e predições

```
    ax.scatter(xc1[:,0], xc1[:,1], color="red", s=10)
    ax.scatter(xc2[:,0], xc2[:,1], color="blue", s=10)
    ax.contour(seqi, seqj, M1, levels=[0], colors="black")
    ax.set_title(f"k={k}, std={std_fixed}")
    ax.set_xlim(0,6)
    ax.set_ylim(0,6)

plt.tight_layout()
plt.show()
```

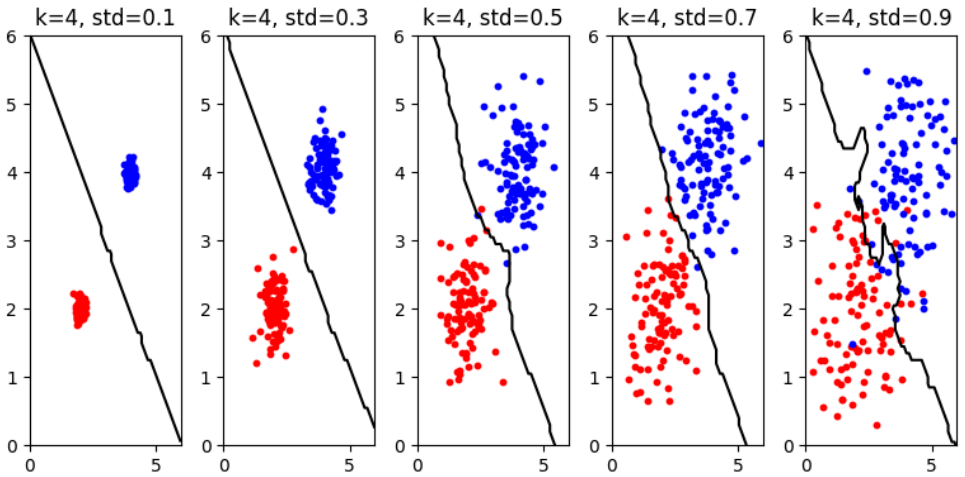
I) Efeito da variação de k (std = 0.95)

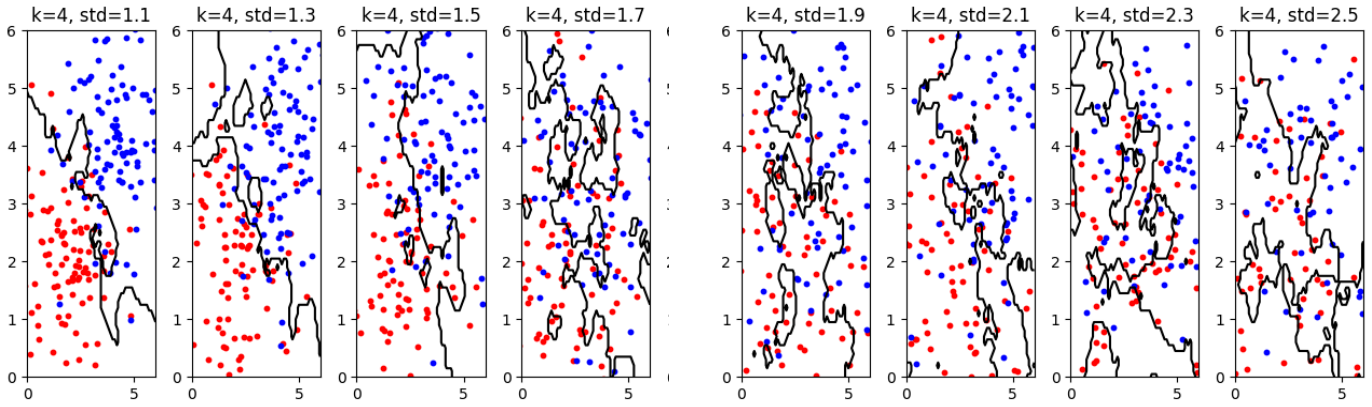
Notamos que o aumento do valor de k leva a uma suavização da fronteira de separação, nos levando a acreditar que um maior valor de k implica em maior precisão do modelo.



II) Efeito da variação do desvio padrão (k = 4)

A variação do desvio padrão das distribuições gaussianas utilizadas no treinamento do modelo nos permite concluir que valores próximos de 0 geram maior overfitting do modelo, uma vez que como os dados não se espalham muito da média eles ficam bastante distantes, gerando uma separação brusca entre classes. Quando o desvio padrão se aproxima de 1, porém se mantém abaixo disso, vemos um nível de espalhamento relevante mas que ainda mantém uma boa precisão na classificação dos dados. Para desvios acima de 1, observamos bastante imprecisão no modelo, em que este se torna ineficiente na separação das classes.

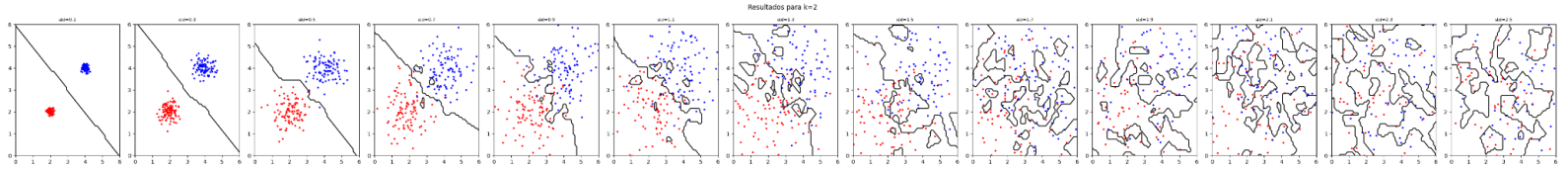




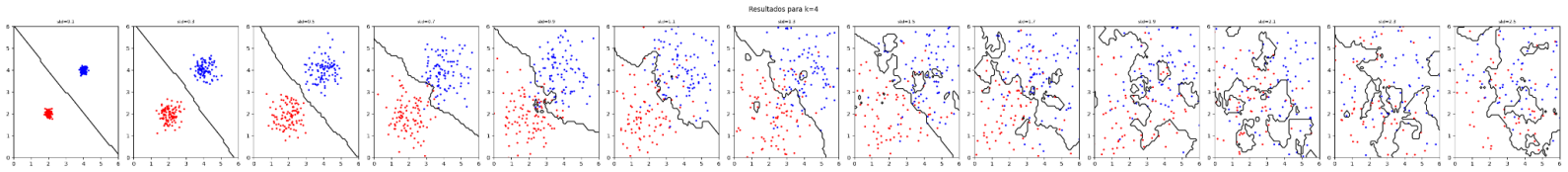
II) Efeito da variação do desvio padrão e de k

A variação de ambos os parâmetros nos permite concluir que o aumento de k leva uma maior tolerância do modelo ao aumento do desvio padrão, ou seja, quanto maior o k maior é o valor de desvio padrão que gera os resultados mais precisos e os problemas de classificação passam a ser postergados para maiores valores de desvio padrão.

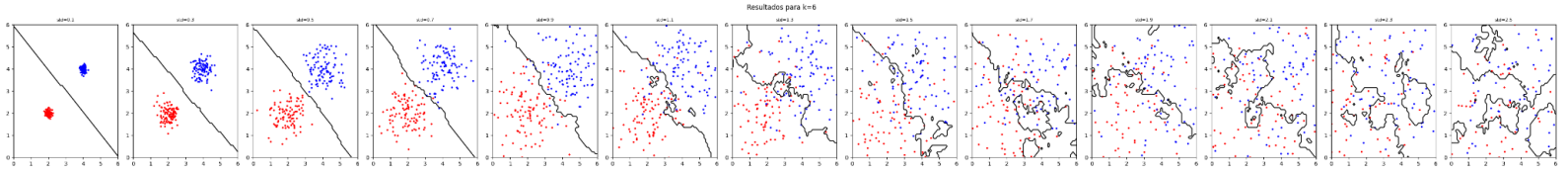
i) k = 2



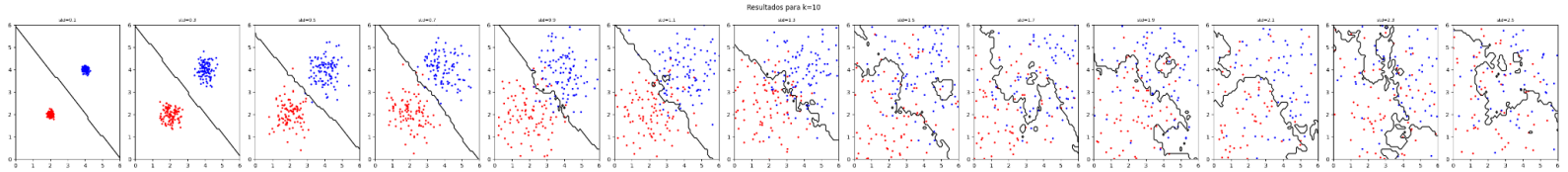
ii) k =4



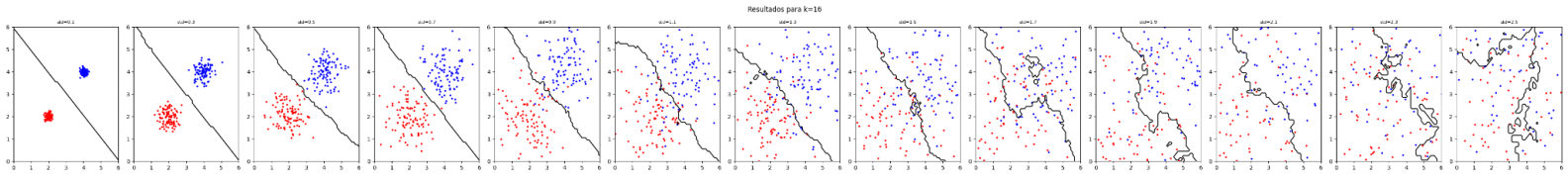
iii) k = 6



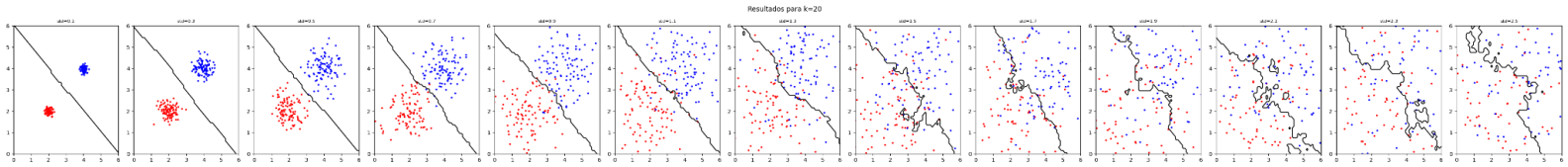
iv) k = 10



v) k = 16



vi) k = 20



Conclusão

O algoritmo do KNN é de fácil implementação e apresenta resultados interessantes para problemas de classificação. Contudo, é possível identificar alta complexidade computacional em sua execução, porém faz-se necessário a comparação deste com outros modelos de classificação para que possamos compreender com mais profundidade os benefícios deste modelo.

Anexos

Full code

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))

def knn(X, Y, xt, k):
    N = len(X)
    distances = []
    for i in range(N):
        d = euclidean_distance(xt, X[i])
        distances.append((d, Y[i]))

    distances.sort(key=lambda x: x[0])
    neighbors = distances[:k]
    classes = [c for _, c in neighbors]

    return Counter(classes).most_common(1)[0][0]

# parâmetros
k_values = range(2,22,2)
std_values = np.arange(0.1, 2.6, 0.2)
nc = 100
seqi = np.arange(0, 6.1, 0.1)
seqj = np.arange(0, 6.1, 0.1)
```

Full code

```
# ----------------------------
# 1) Fixar k, variar std
# ----------------------------
k_fixed = 4
fig, axes = plt.subplots(1, len(std_values), figsize=(20, 4))
for i, std in enumerate(std_values):
    ax = axes[i]
    xc1 = np.random.normal(0, std, (nc, 2)) + np.array([2, 2])
    xc2 = np.random.normal(0, std, (nc, 2)) + np.array([4, 4])
    X = np.vstack((xc1, xc2))
    Y = np.array([1]*nc + [-1]*nc)

    M1 = np.zeros((len(seqi), len(seqj)))
    for ci, xi in enumerate(seqi):
        for cj, yj in enumerate(seqj):
            xt = np.array([xi, yj])
            M1[ci, cj] = knn(X, Y, xt, k_fixed)

    ax.scatter(xc1[:,0], xc1[:,1], color="red", s=10)
    ax.scatter(xc2[:,0], xc2[:,1], color="blue", s=10)
    ax.contour(seqi, seqj, M1, levels=[0], colors="black")
    ax.set_title(f"k={k_fixed}, std={std:.1f}")
    ax.set_xlim(0,6)
    ax.set_ylim(0,6)

plt.tight_layout()
plt.show()
```

Full code

```
# ----------------------------
# 2) Fixar std, variar k
# # ----------------------------
std_fixed = 0.95
fig, axes = plt.subplots(1, len(k_values), figsize=(20, 4))
for i, k in enumerate(k_values):
    ax = axes[i]
    xc1 = np.random.normal(0, std_fixed, (nc, 2)) + np.array([2, 2])
    xc2 = np.random.normal(0, std_fixed, (nc, 2)) + np.array([4, 4])
    X = np.vstack((xc1, xc2))
    Y = np.array([1]*nc + [-1]*nc)

    M1 = np.zeros((len(seqi), len(seqj)))
    for ci, xi in enumerate(seqi):
        for cj, yj in enumerate(seqj):
            xt = np.array([xi, yj])
            M1[ci, cj] = knn(X, Y, xt, k)

    ax.scatter(xc1[:,0], xc1[:,1], color="red", s=10)
    ax.scatter(xc2[:,0], xc2[:,1], color="blue", s=10)
    ax.contour(seqi, seqj, M1, levels=[0], colors="black")
    ax.set_title(f"k={k}, std={std_fixed}")
    ax.set_xlim(0,6)
    ax.set_ylim(0,6)
plt.tight_layout()
plt.show()
```

Full code

```
# ----------------------------
# 3) Variar k e std ao mesmo tempo (grade)
# ----------------------------
for j, k in enumerate(k_values):
    fig, axes = plt.subplots(1, len(std_values), figsize=(3*len(std_values), 4))

    for i, std in enumerate(std_values):
        ax = axes[i] if len(std_values) > 1 else axes

        xc1 = np.random.normal(0, std, (nc, 2)) + np.array([2, 2])
        xc2 = np.random.normal(0, std, (nc, 2)) + np.array([4, 4])
        X = np.vstack((xc1, xc2))
        Y = np.array([1]*nc + [-1]*nc)

        M1 = np.zeros((len(seqi), len(seqj)))
        for ci, xi in enumerate(seqi):
            for cj, yj in enumerate(seqj):
                xt = np.array([xi, yj])
                M1[ci, cj] = knn(X, Y, xt, k)


        ax.scatter(xc1[:,0], xc1[:,1], color="red", s=5)
        ax.scatter(xc2[:,0], xc2[:,1], color="blue", s=5)
        ax.contour(seqi, seqj, M1, levels=[0], colors="black")
        ax.set_title(f"std={std:.1f}", fontsize=8)
        ax.set_xlim(0,6); ax.set_ylim(0,6)

    plt.tight_layout()
    plt.suptitle(f"Resultados para k={k}", fontsize=12, y=1.05)
    plt.show()
```