Métricas em Machine Learning

Classificador baseado no algoritmo de tipo Lloyd para figuras 5×5

Matemática e Computação

Ana Claúdia	Beatriz Marques	Jéssica Gomes	M. Inês Costa	Matilde Domingues
(PG55613)	(PG57743)	(PG55617)	(PG57356)	(A98982)

15 de dezembro de 2024

Conteúdo

Resumo.

Este relatório descreve a implementação de um algoritmo de tipo *Lloyd* para a clusterização de figuras binárias de 5x5 pixels. Utilizando uma métrica de dissimilaridade específica, os eventos foram particionados em três *clusters*. Além disso, foi desenvolvido um classificador para atribuir novos eventos aos *clusters* existentes. A performance do classificador foi avaliada utilizando tabelas de confusão e outras métricas apropriadas. Os resultados obtidos foram analisados e comparados com outras métricas de dissimilaridade.

Capítulo 1

Introdução

Este trabalho tem como objetivo principal proceder à implementação de um algoritmo de tipo *Lloyd* para a clusterização de figuras binárias de 5x5 pixels, utilizando técnicas de aprendizagem não supervisionada. A **clusterização** é uma técnica fundamental em *Machine Learning* (ML) que permite agrupar dados em *clusters* com base numa métrica de dissimilaridade, de forma a identificar padrões e semelhanças intrínsecas nos dados sem depender de dados rotulados pré-definidos.

Neste contexto, utiliza-se uma métrica específica para medir a dissimilaridade entre as figuras, de modo a particioná-las em três *clusters* distintos. O representante de cada *cluster* será determinado com base no conceito de medóide, selecionando o elemento que minimiza uma função custo predefinida. Adicionalmente, desenvolve-se um classificador capaz de atribuir novos eventos aos *clusters* previamente definidos, eliminando a necessidade de reexecutar o algoritmo de Lloyd sempre que novos dados forem introduzidos. A eficácia deste classificador será avaliada utilizando dois conjuntos de dados de teste fornecidos, que incluem a classificação esperada para cada figura. A análise da sua performance será realizada através da construção de tabelas de confusão e do cálculo de métricas apropriadas.

Por fim, este relatório descreve detalhadamente a metodologia a utilizar, a implementação do algoritmo de Lloyd e do classificador, e apresenta uma análise abrangente dos resultados obtidos, destacando os desafios encontrados e as conclusões retiradas.

Capítulo 2

Metodologia

2.1 Algoritmo de *Lloyd*

O algoritmo de *Lloyd* trata-se de um método iterativo de clusterização que visa minimizar a dissimilaridade dentro de cada *cluster*. Neste algoritmo, cada *cluster* é representado por um ponto central chamado centróide. A sua principal característica é a busca por minimizar a soma das distâncias quadradas entre os dados e os centróides, promovendo uma formação de *clusters* compactos e bem definidos.

A execução do algoritmo de *Lloyd* segue dois passos principais, que se repetem até à sua convergência, ou seja, até que não existam mais mudanças nas atribuições dos pontos ou nas posições dos centróides:

- Cada conjunto de dados é associado ao centróide mais próximo, geralmente utilizando a distância euclidiana como métrica de proximidade.
- Após a formação dos *clusters*, os centróides são recalculados através da média aritmética dos pontos pertencentes a cada *cluster*.

2.1.1 *K-Means* vs *K-Medoids*: Diferenças Fundamentais

Embora o *K-Means* se trate de uma aplicação direta do algoritmo de *Lloyd*, o *K-Medoids* surge como uma extensão projetada para superar a sensibilidade do *K-Means* a **outliers**. A seguir, destacam-se as principais diferenças entre as duas abordagens:

- **Representação do *Cluster*:** O *K-Means* calcula centróides como a média dos dados, podendo não corresponder a pontos reais. No *K-Medoids*, os centróides

são obrigatoriamente pontos reais que minimizam a soma das distâncias aos restantes pontos.

- **Sensibilidade a *Outliers*:** O *K-Means* é sensível a valores extremos, que podem distorcer as médias, enquanto que o *K-Medoids* é mais robusto.
- **Complexidade Computacional:** O *K-Means* é mais eficiente devido ao cálculo direto das médias. No entanto, apesar do maior custo, o *K-Medoids* acaba por ser ideal quando a robustez é prioritária.
- **Aplicações:** O *K-Means* é indicado para dados homogéneos e sem valores extremos. O *K-Medoids* adapta-se melhor a cenários com *outliers* ou métricas de distância específicas.

2.1.2 Relevância para o Trabalho

No contexto deste trabalho, o algoritmo de *Lloyd* será empregado como base para o método *K-Medoids*, devido à sua robustez em relação a *outliers* e à capacidade de adaptação a métricas específicas. Neste caso, foi utilizada uma métrica de dissimilaridade baseada na contagem de pixels diferentes entre duas figuras binárias de 5x5 pixels.

A aplicação deste método permitirá identificar padrões e particionar o conjunto de dados de forma eficiente, facilitando as análises subsequentes e o desenvolvimento das etapas seguintes.

2.2 Métrica de Dissimilaridade

A métrica de dissimilaridade utilizada é definida como:

$$d(x, y) = \frac{1}{25} \sum_{i=1}^{25} d_i(x, y)$$

onde $d_i(x, y) = 1$ se $x_i \neq y_i$ e $d_i(x, y) = 0$ se $x_i = y_i$.

2.3 Escolha do Medóide

O representante de cada *cluster*, ou medóide, é o elemento que minimiza a função de custo dentro do *cluster*:

$$E(m; C) = \sum_{x \in C} d(m, x)$$

A função de custo é calculada para cada elemento do *cluster* e o elemento com o menor custo é escolhido como o medóide.

2.4 Implementação do Classificador

Após a clusterização inicial, implementamos então um classificador para atribuir novos eventos aos *clusters* existentes. O classificador utiliza a partição obtida pelo algoritmo de *Lloyd* e atribui novos eventos ao *cluster* mais próximo com base na métrica de dissimilaridade, sem necessidade de voltar a realizar o algoritmo.

Capítulo 3

Implementação

3.1 Dados Utilizados

Os dados utilizados neste trabalho foram fornecidos nos arquivos *BD1.txt*, *teste.txt* e *teste2.txt*. Cada linha dos arquivos contém os valores dos 25 pixels de uma figura binária. O 26º valor dos arquivos de teste representa a classe da figura.

Analisando os ficheiros através do código MATLAB (disponível na sua totalidade em `READ_DATA.m`), foi possível transformar cada linha do ficheiro na sua respetiva imagem 5x5:

```
for i = 1:n_rows
    % Reshape da linha atual para uma matriz 5x5
    img = reshape(testData(i, :), [5, 5]);

    % Girar a imagem 90 graus no sentido horário
    img_rotated = rot90(img, 1); % -1 para
    % sentido horário (1 para sentido anti-horário)

    % Mostrar a imagem
    figure; % Abre nova figura
    imagesc(img_rotated); % Mostra a matriz como imagem
    colormap(custom_colormap); % Aplica as cores personalizadas
    colorbar; % Exibe a barra de cores
    title(['Imagem ', num2str(i)]);
end
```

Foi possível identificar que os padrões das figuras se assemelhavam a duas letras (A e P) e ao símbolo +:

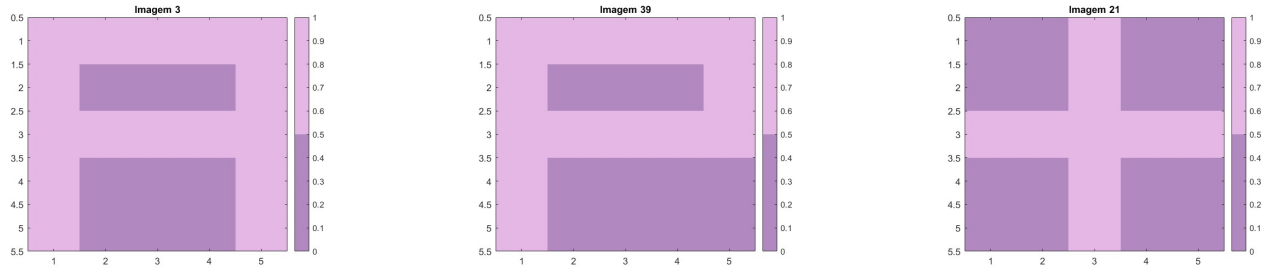

Figura 3.1: Padrões identificados numa análise inicial de *BD1.txt*, *teste.txt* e *teste2.txt*

3.2 Algoritmo de *Lloyd*

A implementação do algoritmo de *Lloyd* foi realizada utilizando MATLAB, com o objetivo de particionar as figuras binárias presentes no arquivo *BD1.txt* em três *clusters* distintos. Para alcançar esse objetivo, foram seguidos os passos descritos na **Secção 2.1**, que detalham a metodologia aplicada.

```
I = 25; % cada linha tem 25 valores, onde cada uma representa um
% valor armazenado na matriz x

% Definir o número de clusters (k = 3)
K = 3;

% Criar os K representantes aleatórios
i = 0;

while ~feof(data)
    i = i + 1;
    xx = fscanf(data, '%f\n', I); % ler as linhas da BD
    for j = 1:I
        x(i, j) = xx(j);
    end
end

fclose(data);
N = i

% Escolher os 3 representantes iniciais
antigos_representantes = [1, 12, 23]; % escolher, aleatoriamente, 3
% eventos como representantes iniciais
```

(continuação)

```
% Exibir os números gerados
CP = 1;
it = 0;

while  CP > 0.0001 && it < 10
 P = calcula_particao(antigos_representantes, x, K);
 % calcular uma particao usando os representantes anteriores através da
 % dissimilaridade de cada evento em relacao aos representantes

 novos_representantes = calcula_representantes(P, x, N, K);
 % calcular novos representantes com base nos eventos atribuídos a cada
 % cluster, minimizando o custo da dissimilaridade

 CP = diferenca_representantes(antigos_representantes, ...
    novos_representantes);
 % medida da diferenca entre os novos representantes e os antigos,
 % verificando se o algoritmo convergiu

 antigos_representantes = novos_representantes;
 it = it + 1;
end
```

De forma a melhorar a organização e a legibilidade do código, várias funções auxiliares foram criadas para desempenhar tarefas específicas dentro do processo (nomeadamente funções para construir e calcular partições, calcular e imprimir representantes, calcular a diferença entre dois representantes, aplicar a função de dissemelhança e calcular o custo). Estas encontram-se definidas no **Apêndice A**.

O código principal do algoritmo de *Lloyd* encontra-se no ficheiro `principal.mat`, onde o fluxo do algoritmo é estruturado e as operações de clusterização são realizadas.

3.3 Classificador

Como referido na **Secção 2.3**, o classificador foi desenvolvido para atribuir novos eventos aos *clusters* existentes sem a necessidade de reexecutar o algoritmo de clusterização. A performance do classificador foi avaliada utilizando os dados dos arquivos *teste.txt* e *teste2.txt*.

```
% Carregar as matrizes guardadas no principal.m
load('principal.mat');

% Teste
I = 25; % Número de características por evento
i = 1;
teste2 = fopen('teste2.txt', 'r'); % Ler o ficheiro de teste que
% contém linhas com 26 números alterar para teste.m se quisermos
% analisar esses resultados

while ~feof(teste2)
    yk = fscanf(teste2, '%f\n', (I + 1)); % Ler as linhas da BD

    % Leitura da vigésima-quinta posicao em xx e da vigésima-sexta ...
       posicao em y
    yy = yk(1:25);
    yyy = yk(26);
    for j = 1:I
        y(i, j) = yy(j);
    end
    v(1,i) = yyy(1);
    i = i + 1;
end

fclose(teste2);

for i = 1:length(y)
    d1 = calcula_dissemelhanca(x(novos_representantes(1), :), y(i, :));
    d2 = calcula_dissemelhanca(x(novos_representantes(2), :), y(i, :));
    d3 = calcula_dissemelhanca(x(novos_representantes(3), :), y(i, :));
    % Calcular a dissimilaridade com cada um dos tres representantes
    d = [d1 d2 d3];
    [mn, id] = min(d);
    Q(i) = id(1); % Escolher o representante com menor dissimilaridade
    % e atribuir o índice do cluster correspondente em Q
end

% Definir as cores personalizadas:
custom_colormap = [178/255, 136/255, 192/255;
                   228/255, 183/255, 229/255];

% Loop para exibir gráficos para cada representante
imprimir_representantes(novos_representantes, x);
```

(continuação)

```
colormap(custom_colormap);
for i = 1:60 % Este ciclo exibe matrizes 5x5 rodadas para cada evento
    % Criar uma nova figura para cada linha
    figure;

    % Reshape da linha atual para formar uma matriz 5x5
    matriz_5x5 = reshape(y(i, :), 5, 5);

    % Rodar a matriz em 90 graus para a direita - a rotacao  melhora
    % a visao
    matriz_rodada = imrotate(matriz_5x5, 90);

    % Exibir a matriz usando a funcao imagesc
    imagesc(matriz_rodada);

    % Adicionar rótulos e título
    xlabel('Linha');
    ylabel('Coluna');  % Inversao dos rótulos para refletir a rotacao
    title(['Linha ' num2str(i)]);

    % Adicionar barra de cores
    colorbar;
end

% Exibir a matriz de confusao
confMat = confusionchart(v, Q); % confusionchart para as classificacoes
% reais com as previstas (Q)

% Calcular a precisao
accuracy = sum(diag(confMat.NormalizedValues)) / ...
   sum(confMat.NormalizedValues(:));
% Calcula a precisao usando a diagonal normalizada da matriz confusao

% Adicionar a precisao do modelo ao título
title(['Matriz de Confusao (Precisao: ' num2str(accuracy*100) '%)']);

Q
```

Capítulo 4

Resultados Obtidos

4.1 Representantes Obtidos

Em baixo encontram-se os 3 representantes obtidos para a métrica e classificadores utilizados.

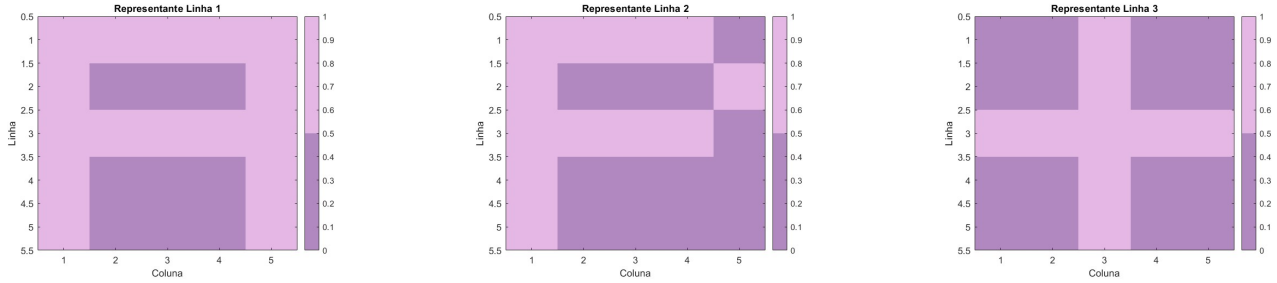

Figura 4.1: Representantes obtidos para *teste.txt* e *teste2.txt*

4.2 Matriz de Confusão para o Arquivo teste.txt

A matriz de confusão obtida ao avaliar o classificador com os dados presentes no arquivo *teste.txt* apresenta uma precisão de **98,333%**. Esta alta precisão reflete o bom desempenho do classificador em atribuir corretamente os eventos aos *clusters* predefinidos. Apresentamos a matriz de confusão correspondente:

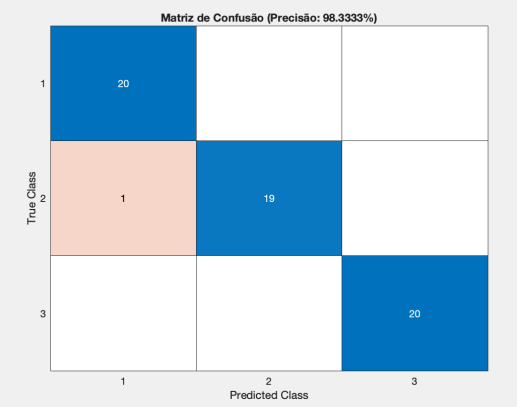

Figura 4.2: Matriz de confusão da análise de *teste.txt*

Dos 20 eventos reais da classe 1 e 3, todos foram classificados corretamente, evidenciando a confiabilidade do classificador para estas classes.
Relativamente à classe 2, apenas 1 evento foi erroneamente classificado como pertencente à classe 1, o que representa um erro menor no contexto geral.

4.3 Matriz de Confusão para o Arquivo teste2.txt

Ao utilizar os dados do arquivo *teste2.txt*, a precisão do classificador baixou para **80%**. A matriz de confusão correspondente está apresentada abaixo:

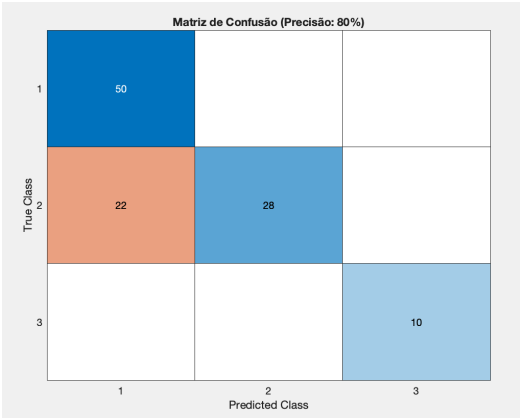

Figura 4.3: Matriz de confusão da análise de *teste2.txt*

Analisando ambas as matrizes de confusão, todos os 50 eventos reais da classe 1 foram classificados corretamente, demonstrando um desempenho consistente para esta classe.
Dos 50 eventos reais da classe 2, apenas 28 foram classificados corretamente, enquanto 22 foram atribuídos à classe 1, de forma errada. Esse comportamento reduz a performance geral do classificador.
Dos 10 eventos reais da classe 3, todos foram classificados corretamente, indicando

robustez do classificador para esta classe, apesar de termos poucas amostras.

4.3.1 Observações sobre os dados

Algumas diferenças apresentadas nos dois ficheiros podem justificar a diferença de precisão nas matrizes de confusão:

- Em *teste.txt*, as classes parecem estar bem separadas, o que facilita a sua classificação. Já em *teste2.txt*, há uma maior sobreposição, especialmente entre as classes 1 e 2, evidenciado pelo número elevado de eventos da classe 2 que são classificados de forma errada como classe 1.
- Alguns eventos da classe 2 no *teste2.txt* têm características muito parecidas com os da classe 1, o que leva a mais confusões.
- Em *teste2.txt*, a classe 3 tem menos amostras comparadas às classes 1 e 2, o que pode influenciar a capacidade do classificador de generalizar corretamente para os dados de teste.

Na figura seguinte segue o exemplo de um evento de *teste2.txt* de classe 1, cujas características são bastante semelhantes com a classe 2, levando a um erro de classificação:

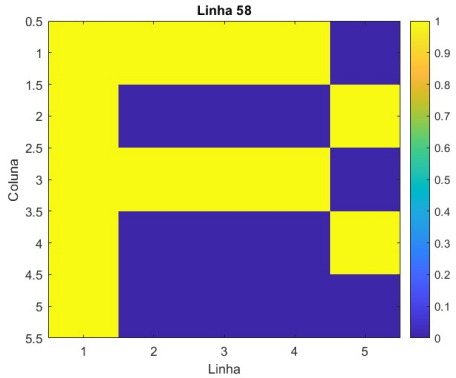

Figura 4.4: Exemplo de um evento de *teste2.txt* de classe 1.

Capítulo 5

Conclusão

Este trabalho apresentou a implementação de um algoritmo de tipo *Lloyd* aplicado à clusterização de figuras binárias de 5x5 pixels, utilizando a abordagem de *K-Medoids*. A metodologia desenvolvida demonstrou a eficácia do algoritmo ao particionar os dados em *clusters* distintos e criar um classificador capaz de atribuir novos eventos com base nas partições obtidas.

Os resultados evidenciaram o alto desempenho do classificador no arquivo *teste.txt*, com uma precisão de **98,333%**. No entanto, a análise do arquivo *teste2.txt* revelou limitações na capacidade de generalização do classificador, devido à maior sobreposição entre as classes e à complexidade dos padrões. Essa diferença de desempenho ressalta a importância de avaliar cuidadosamente a métrica de dissimilaridade e considerar ajustes no algoritmo para lidar com dados mais desafiadores.

Como trabalho futuro, sugere-se:

- Refinar a métrica de dissimilaridade para melhorar a separação entre classes próximas.
- Explorar outras estratégias de inicialização dos medóides para evitar possíveis convergências locais inadequadas.
- Avaliar o impacto de algoritmos alternativos, em cenários com maior sobreposição entre classes.

Em conclusão, o trabalho contribuiu para a compreensão e aplicação prática do algoritmo de *Lloyd* num cenário de clusterização e classificação, identificando as suas potencialidades e limitações face a diferentes características dos dados.

Bibliografia

[1] Seung-Seok Choi, Sung-Hyuk Cha, Charles C. Tappert. *A Survey of Binary Similarity and Distance Measures*, Systemics, Cybernetics and Informatics, Vol. 8(1), 2010.

[2] O. Simeoni. *A Brief Introduction to Machine Learning for Engineers*, Online: https://arxiv.org/pdf/1709.02840.pdf, 2018.

[3] Christopher De Sa. *Notes of Cornell University on Introduction to Machine Learning Lecture 4*, Online: https://www.cs.cornell.edu/courses/cs4780/2022sp/corereferences, 2022.

Apêndice A

Códigos da Implementação no Matlab

READ_DATA.m

```
% Carregar os dados do arquivo
testData = readmatrix(['teste.txt']);  % Le o arquivo

% Verificar o tamanho dos dados
[n_rows, n_cols] = size(testData);

% Remover a última coluna
testData = testData(:, 1:25);

% Definir as cores personalizadas:
custom_colormap = [178/255, 136/255, 192/255;  % African Violet (B288C0)
                   228/255, 183/255, 229/255]; % Pink Lavender (E4B7E5)

for i = 1:n_rows
    % Reshape da linha atual para uma matriz 5x5
    img = reshape(testData(i, :), [5, 5]);

    % Girar a imagem 90 graus no sentido horário
    img_rotated = rot90(img, 1); % -1 para sentido horário (1 para ...
       sentido anti-horário)

    % Mostrar a imagem
    figure; % Abre nova figura
    imagesc(img_rotated); % Mostra a matriz como imagem
    colormap(custom_colormap); % Aplica as cores personalizadas
    colorbar; % Exibe a barra de cores
    title(['Imagem ', num2str(i)]);
end
```

principal.m

```
clear all
clc

% OBJETIVO: Processar uma base de dados 'BD1.txt' para agrupar
% eventos em clusters, bem como determinar representantes iniciais,
% calcular particoes e ajustar os representantes

% Passo 1: Ler os dados do arquivo
data = fopen('BD1.txt','r'); % o ficheiro deve estar na mesma pasta
I = 25; % cada linha tem 25 valores, onde cada uma representa um
% valor armazenado na matriz x

% Inicializar a matriz x
x = zeros(1000, I);  % Suponha que há 1000 linhas na base de dados,
% ajustável conforme necessário

i = 0;
while ~feof(data)
    i = i + 1;
    xx = fscanf(data, '\%f\n', I); % ler as linhas da BD
    x(i, :) = xx; % Preencher a linha i da matriz x com os dados lidos
end

fclose(data);
N = i; % Número de elementos lidos da base de dados

% Escolher os 3 representantes iniciais
antigos_representantes = [1, 12, 23];
% escolher, aleatoriamente, 3 eventos como representantes iniciais
```

(continuação)

```
CP = 1; % Exibir os números gerados
it = 0;

while  CP > 0.0001 && it < 10
    P = calcula_particao(antigos_representantes, x, K);
    % calcular uma particao usando os representantes anteriores
    % através da dissimilaridade de cada evento em relacao
    % aos representantes

    novos_representantes = calcula_representantes(P, x, N, K);
    % calcular novos representantes com base nos eventos atribuídos a
    % cada cluster, minimizando o custo da dissimilaridade

    CP = diferenca_representantes(antigos_representantes, ...
       novos_representantes);
    % medida da diferenca entre os novos representantes e os antigos,
    % verificando se o algoritmo convergiu

    antigos_representantes = novos_representantes;
    it = it + 1;
end

save('principal.mat', 'novos_representantes', 'x'); % guardar os
% representantes finais e os dados num arquivo para uso posterior
```

teste.m

```
clear all
clc

% Carregar as matrizes guardadas no principal.m
load('principal.mat')

% Teste
I = 25; % Número de características por evento
i = 1;
teste2 = fopen('teste2.txt', 'r'); % Ler o ficheiro de teste que
% contém linhas com 26 números
```

(continuação)

```
while ~feof(teste2)
    yk = fscanf(teste2, '%f\n', (I + 1)); % Ler as linhas da BD

    % Leitura da vigésima-quinta posicao em xx e da vigésima-sexta ...
       posicao em y
    yy = yk(1:25);
    yyy = yk(26);
    for j = 1:I
        y(i, j) = yy(j);
    end
    v(1,i) = yyy(1);
    i = i + 1;
end

fclose(teste2);


for i = 1:length(y)
    d1 = calcula_dissemelhanca(x(novos_representantes(1), :), y(i, :));
    d2 = calcula_dissemelhanca(x(novos_representantes(2), :), y(i, :));
    d3 = calcula_dissemelhanca(x(novos_representantes(3), :), y(i, :));
    % Calcular a dissimilaridade com cada um dos tres representantes

    d = [d1 d2 d3];

    [mn, id] = min(d);
    Q(i) = id(1); % Escolher o representante com menor dissimilaridade
    % e atribuir o índice do cluster correspondente em Q
end


% Loop para exibir gráficos para cada representante
imprimir_representantes(novos_representantes, x);

for i = 1:60 % Este ciclo exibe matrizes 5x5 rodadas para cada evento
    % Criar uma nova figura para cada linha
    figure;

    % Reshape da linha atual para formar uma matriz 5x5
    matriz_5x5 = reshape(y(i, :), 5, 5);

    % Rodar a matriz em 90 graus para a direita - a rotacao  melhora
    % a visao
    matriz_rodada = imrotate(matriz_5x5, 90);
```

(continuação)

```
    % Exibir a matriz usando a funcao imagesc
    imagesc(matriz_rodada);

    % Adicionar rótulos e título
    xlabel('Linha');
    ylabel('Coluna');  % Inversao dos rótulos para refletir a rotacao
    title(['Linha ' num2str(i)]);

    % Adicionar barra de cores
    colorbar;
end

% Exibir a matriz de confusao
confMat = confusionchart(v, Q); % confusionchart para as classificacoes
% reais com as previstas (Q)

% Calcular a precisao
accuracy = sum(diag(confMat.NormalizedValues)) / ...
   sum(confMat.NormalizedValues(:));
% Calcula a precisao usando a diagonal normalizada da matriz confusao

% Adicionar a precisao do modelo ao título
title(['Matriz de Confusao (Precisao: ' num2str(accuracy*100) '%)']);

Q
```

imprimir_representantes

```
function i = imprimir_representantes(elemento, x) % esta funcao exibe
% gráficos 5x5 de cada representante

    for i = 1:length(elemento)

        % Criar uma nova figura para cada linha
        figure;

        % Reshape da linha atual para formar uma matriz 5x5
        matriz_5x5 = reshape(x(elemento(i), :), 5, 5);
        matriz_5x5 = imrotate(matriz_5x5, 90);

        % Exibir a matriz usando a funcao imagesc
        imagesc(matriz_5x5);

        % Adicionar rótulos e título
        xlabel('Coluna');
        ylabel('Linha');
        title(['Representante Linha ' num2str(i)]);
        x(elemento(i), :)

        % Adicionar barra de cores
        colorbar;
    end
end
```

diferenca_representantes

```
function CP = diferenca_representantes(antigos_representantes, ...
   novos_representantes)
% Verifica a variacao das coordenadas dos representantes inicias com
% os novos representantes

CP = sum(antigos_representantes ~= novos_representantes);

end
```

constroi_particao

```
function P = constroi_particao(d, N, K)
% d(i,j) - tem distancia de evento i ao representante j;
% N é número de eventos;
% K é número de representantes.

% para todos os eventos e todos os representantes, ve qual é a distancia
% mínima do evento ao representante e atribui a posicao i da particao o
% índice do representante mais próximo

  for j = 1:N
        [ord, ind] = sort([d(1, j), d(2, j), d(3, j)]);
        P(j) = ind(1);
  end

end
```

calcula_representantes

```
function novos_representantes = calcula_representantes(P, x, N, K)
% calcula novos representantes com base nos eventos atribuídos a cada
% cluster, minimizando o custo de dissimilaridade

novos_representantes = zeros(1, K);  % Cria um vetor para armazenar
% o índice de cada novo representante dos clusters.
% Inicialmente, todos os valores sao zero

for i = 1:K
    % Encontrar os índices dos eventos pertencentes a classe i (ou
    % seja, aqueles que foram atribuídos a este cluster na particao P)
    indices_classe = find(P == i);

    custos = zeros(1, length(indices_classe));  % Cria um vetor para
    % armazenar os custos de dissimilaridade para cada evento no
    % cluster i

    % Calcula o custo para cada elemento ser representante da classe i
    for j = 1:length(indices_classe)
        custos(j) = calcula_custo(x(indices_classe(j), :), ...
           x(indices_classe, :));
    end
```

(continuação)

```
    % Encontra o elemento com o menor custo
    [ord, indice_menor_custo] = min(custos);

    % Armazena o índice do elemento com menor custo como novo
    % representante
    novos_representantes(i) = indices_classe(indice_menor_custo);

end

end
```

calcula_particao

```
function P = calcula_particao(linha_do_representante, x, K)
% determinar a particao (P) atribuindo cada evento ao representante
% mais próximo, com base nas dissimilaridades calculadas

    N = length(x); % Número de elementos da BD

    % Inicializa a matriz de dissimilaridades
    d = zeros(K, N);

    % Para todos os eventos, calcula a dissimilaridade entre o
    %evento e os representantes
    for i = 1:K % para todos os eventos
        for j = 1:N % para todos os clusters

            % d(i,j) é a dissimilaridade do evento i ao representante j
            d(i,j) = ...
               calcula_dissemelhanca(x(linha_do_representante(i), :), ...
               x(j, :)); % calcula a dissimilaridade entre o i-ésimo
            % representante do cluster e o j-ésimo evento da base de
            % dados. Esta medida é usada para determinar o cluster
            % mais próximo de cada evento e ajustar os representantes

        end
    end

    % Exibe a matriz de dissimilaridades
    disp('Matriz de Dissimilaridades:');
    disp(d);
```

(continuação)

```
    % Constrói a particao com base nas dissimilaridades
    P = constroi_particao(d, N, K); % P contém lista de índices dos
    % representantes para cada evento da BD

end
```

calcula_dissemelhanca

```
function d = calcula_dissemelhanca(x_R, x_i) % calcular diferencas
%elementares entre dois vetores x_R e x_i, ou seja, que mede o quao
%"diferentes" dois vetores (eventos) sao

    d = sum(x_R ~= x_i);
    f = x_i(5) + x_i(10) + x_i(15) + x_i(20);
    s=  x_R(5) + x_R(10) + x_R(15) + x_R(20);
    if f <= s
        d = d + 2;
    end
    d = 0.04*(d - sum(x_R & x_i))
end
```

calcula_custo

```
function custo = calcula_custo(elemento, elementos_classe) % Soma
% das dissimilaridades entre um elemento e todos os outros elementos
% de sua classe

    % Inicializa o custo
    custo = 0;

    % Calcula o custo como a soma das dissimilaridades do elemento
    % com todos os outros da classe
    for k = 1:size(elementos_classe, 1)
        custo = custo + calcula_dissemelhanca(elemento, ...
           elementos_classe(k, :));
    end

end
```