Algoritmos de Agrupamento Particionado Rígido Baseados em Múltiplas Matrizes de Dissimilaridade

O agrupamento particionado rígido organiza conjuntos de objetos em clusters de forma que os objetos de um mesmo cluster são altamente semelhantes entre si, enquanto os objetos de clusters diferentes são altamente distintos. A proposta deste artigo expande este conceito ao lidar com cenários nos quais os dados são descritos por múltiplas matrizes de dissimilaridade, permitindo uma análise mais robusta e flexível. Ao integrar diferentes funções de dissimilaridade ou variáveis, obtemos representações complementares dos dados, o que permite uma melhor segmentação. O artigo apresenta uma extensão do algoritmo SRDCA, permitindo o uso simultâneo de múltiplas matrizes de dissimilaridade e a introdução de pesos de relevância ajustados local ou globalmente para cada matriz de dissimilaridade.

Metodologia

Representação dos Dados:

Os dados a serem agrupados são representados por matrizes de dissimilaridade D=[d(ei,el)]D = [d(e_i, e_l)], onde d(ei,el)d(e_i, e_l) representa a dissimilaridade entre os objetos eie_i e ele_l. Cada matriz de dissimilaridade pode ser derivada de diferentes funções ou variáveis, capturando aspectos distintos dos dados.

Algoritmos Propostos:

1. **MRDCA (Multiple Dissimilarity Matrices Clustering Algorithm)**:
    - Esta é uma extensão do algoritmo SRDCA para múltiplas matrizes de dissimilaridade, assumindo uma relevância igual para todas as matrizes. O algoritmo tenta combinar as informações contidas nas diferentes matrizes para criar clusters mais robustos.
2. **MRDCA-RWL (Multiple Dissimilarity Matrices Clustering Algorithm with Relevance Weight Localized)**:
    - Neste algoritmo, os pesos de relevância para cada matriz de dissimilaridade são estimados localmente, ou seja, dentro de cada cluster. Esses pesos podem variar de cluster para cluster e são ajustados iterativamente, o que permite uma adaptação mais precisa às características específicas de cada grupo de dados.
3. **MRDCA-RWG (Multiple Dissimilarity Matrices Clustering Algorithm with Relevance Weight Globalized)**:
    - Ao contrário do MRDCA-RWL, os pesos de relevância neste algoritmo são estimados globalmente e aplicados uniformemente a todos os clusters, simplificando o modelo, mas sacrificando parte da flexibilidade.

Esses algoritmos otimizam um critério de adequação JJ, que mede a correspondência entre os clusters e seus protótipos, levando em consideração tanto as múltiplas matrizes de dissimilaridade quanto os pesos de relevância.

Resultados Experimentais

Conjuntos de Dados Sintéticos:

- **Cenários**: Dados foram gerados a partir de distribuições normais bivariadas, com diferentes variâncias para as variáveis.
- **Algoritmos Avaliados**: SRDCA, MRDCA, MRDCA-RWL, MRDCA-RWG, e o fuzzy CARD-R.
- **Métricas de Desempenho**: Índice Rand Corrigido (CR), Medida F e Taxa de Erro de Classificação (OERC).
- **Conclusões**:
    - **MRDCA-RWL** se mostrou superior em cenários onde as variâncias das variáveis diferiam significativamente, já que ele pode ajustar a relevância de cada matriz de dissimilaridade de acordo com as características do cluster.
    - **MRDCA-RWG** teve melhor desempenho quando as variâncias eram homogêneas, já que um único peso global para todas as matrizes é mais adequado em tais cenários.
    - **NERF** e **SRDCA** se saíram melhor em situações com baixa variabilidade entre as variáveis, onde as matrizes de dissimilaridade não precisam ser tão adaptativas.

Conjuntos de Dados Reais (UCI):

- **Conjuntos de Dados**: Abalone, Iris, Vinho, entre outros.
- **Configuração**: Matrizes de dissimilaridade foram derivadas de atributos individuais e combinações entre atributos.
- **Resultados**: Os experimentos com dados reais mostraram resultados consistentes com os experimentos sintéticos, destacando a eficácia do **MRDCA-RWL** em cenários mais complexos.

Contribuições e Conclusões:

1. **Inovação**: A introdução de relevância adaptativa para múltiplas matrizes de dissimilaridade permite uma análise mais precisa em cenários com múltiplas representações dos dados.
2. **Resultados Consistentes**: O algoritmo **MRDCA-RWL** mostrou-se particularmente útil para dados heterogêneos, enquanto o **MRDCA-RWG** é mais indicado para dados homogêneos.
3. **Trabalhos Futuros**: A aplicação de **agrupamento fuzzy** e a análise de **dados de alta dimensionalidade** são sugeridas como áreas para investigação futura.

Cálculo dos Pesos de Relevância (λkj\lambda_{kj}) no MRDCA-RWL:

O algoritmo MRDCA-RWL foi projetado para realizar o agrupamento considerando múltiplas matrizes de dissimilaridade, estimando os **pesos de relevância** (λkj\lambda_{kj}) para cada matriz de dissimilaridade em cada cluster. Esses pesos são ajustados iterativamente para refletir a importância relativa de cada matriz para a definição do cluster.

Estrutura do MRDCA-RWL:

O algoritmo segue uma abordagem iterativa dividida em três etapas principais:

1. **Cálculo dos protótipos dos clusters**.
2. **Cálculo dos pesos de relevância (λkj\lambda_{kj}) para cada matriz em cada cluster**.

3. **Atualização da partição dos clusters**.

O critério de adequação JJ utilizado pelo algoritmo é definido como:

$$J = \sum_{k=1}^{K} \sum_{e_i \in C_k} \sum_{j=1}^{p} \lambda_{kj} d_j(e_i, G_k)$$

Onde:

- C_k representa o k-ésimo cluster.
- G_k é o protótipo do k-ésimo cluster.
- λ_{kj} é o peso de relevância para a matriz de dissimilaridade D_j no cluster C_k.

Cálculo de λ_{kj}:

O cálculo dos pesos de relevância λ_{kj} é central para o algoritmo. Ele ocorre na segunda etapa, sendo ajustado iterativamente para minimizar o critério JJ. O peso λ_{kj} é encontrado através de uma otimização que deve satisfazer as seguintes restrições:

1. $\lambda_{kj} > 0$ para $j = 1, \dots, p$.
2. $\prod_{j=1}^{p} \lambda_{kj} = 1$, garantindo que os pesos sejam normalizados multiplicativamente.

A fórmula para calcular λ_{kj} é dada por:

$$\lambda_{kj} = \frac{\sum_{e_i \in C_k} \sum_{e \in G_k} d_j(e_i, e)}{\left(\prod_{h=1}^{p} \sum_{e_i \in C_k} \sum_{e \in G_k} d_h(e_i, e)\right)^{1/p}}$$

Onde:

- O numerador é a soma das dissimilaridades ponderadas entre os objetos e_i e o protótipo G_k usando a matriz de dissimilaridade D_j.
- O denominador é a soma ponderada das dissimilaridades de todas as matrizes D_h, usada para balancear a contribuição relativa de cada matriz.

Intuição do Cálculo de λ_{kj}:

Os pesos λ_{kj} refletem a importância relativa de cada matriz de dissimilaridade D_j no cluster C_k. Um valor mais alto de λ_{kj} indica que a matriz D_j é mais relevante para a descrição dos objetos no cluster C_k. A normalização multiplicativa garante que os pesos sejam balanceados entre as diferentes matrizes.

Demonstração Matemática:

Considere um conjunto de dados E={e1,e2,e3,e4}E = \{ e_1, e_2, e_3, e_4 \} e duas matrizes de dissimilaridade D1D_1 e D2D_2 descritas por:

D1=[0123102222013210]D_1 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 2 & 2 \\ 2 & 2 & 0 & 1 \\ 3 & 2 & 1 & 0 \\ \end{bmatrix} D2=[0214203313024320]D_2 = \begin{bmatrix} 0 & 2 & 1 & 4 \\ 2 & 0 & 3 & 3 \\ 1 & 3 & 0 & 2 \\ 4 & 3 & 2 & 0 \\ \end{bmatrix}

Suponha que queremos dividir os dados em dois clusters C1C_1 e (C_2

), com K=2K = 2.

1. **Inicialização**:
    - Escolha aleatória dos protótipos G1={e1}G_1 = \{ e_1 \}, G2={e4}G_2 = \{ e_4 \}.
    - Inicialização dos pesos λ11=λ12=1\lambda_{11} = \lambda_{12} = 1 e λ21=λ22=1\lambda_{21} = \lambda_{22} = 1.
2. **Etapas**:
    - **Atualização dos protótipos**:
        - Recalcular os protótipos G1G_1 e G2G_2 minimizando o critério JJ.
    - **Atualização de λkj\lambda_{kj}**:
        - Usando a fórmula descrita para calcular os pesos de relevância λkj\lambda_{kj} para cada cluster e matriz de dissimilaridade.

Explicação Completa do Código: Implementação do Algoritmo de K-Means Multi-Visão

O código implementa o **algoritmo de k-means** modificado para trabalhar com **duas visões de dados**. O objetivo é segmentar um conjunto de dados em clusters, utilizando as dissimilaridades calculadas a partir de duas representações diferentes dos dados. Isso é feito iterativamente, ajustando os **protótipos de clusters** e a **atribuição dos pontos aos clusters**.

Bibliotecas Importadas:

```
#include <iostream>   // Para entrada e saída de dados.
#include <vector>     // Para utilizar a estrutura de dados 'vector'.
#include <algorithm>  // Para funções como std::max_element, necessária para
encontrar o valor máximo.
#include <iterator>   // Para std::distance, que calcula a distância entre
iteradores.
#include <limits>     // Para std::numeric_limits, utilizado para definir
valores máximos e garantir inicializações adequadas.
```

- **`<iostream>`**: Usado para imprimir dados no console, como os resultados das iterações e a atribuição de clusters.
- **`<vector>`**: Para armazenar os dados de forma dinâmica e utilizar vetores (listas de tamanho variável).
- **`<algorithm>`**: Inclui funções como `std::max_element` que é usada para encontrar o maior protótipo de cluster.
- **`<iterator>`**: Inclui a função `std::distance` que calcula a distância entre dois iteradores.
- **`<limits>`**: Fornece acesso a valores de limites numéricos, como o valor máximo possível para um tipo de dado (usado aqui para inicializar as distâncias como infinito).

Função `multiViewKMeans`:

Essa função é a base do algoritmo de **k-means**, adaptado para trabalhar com duas visões de dados (dois conjuntos de matrizes de dissimilaridade). O algoritmo de k-means é baseado em três etapas principais:

1. **Atribuição de pontos aos clusters**.
2. **Cálculo dos novos protótipos (centros dos clusters)**.
3. **Atualização da partição dos dados (reestruturação dos clusters)**.

Parâmetros da Função:

- **`view1`** e **`view2`**: Duas representações dos dados. Cada uma é um vetor 2D, onde cada linha representa um ponto e as colunas representam características (ou variáveis) dos dados.
- **`k`**: O número de clusters a serem gerados.

Processo do Algoritmo:

1. **Inicialização**:
    - O número de dados, nn, é determinado com base no tamanho de `view1`.
    - Os clusters são inicializados com todos os pontos pertencendo ao cluster 0.
    - Os protótipos dos clusters são inicializados com os primeiros pontos de `view1` e `view2`.

```
int n = view1.size(); // Número de dados
std::vector<int> clusters(n, 0); // Atribuindo todos inicialmente ao cluster 0
std::vector<std::vector<double>> prototypes(k,
std::vector<double>(view1[0].size(), 0));
prototypes[0] = view1[0];
prototypes[1] = view2[0];
```

2. **Iteração Principal** (10 iterações):
    - O loop de iteração começa, com 10 iterações definidas para realizar a atualização dos clusters e protótipos.
3. **Passo 1 - Atribuição de Pontos aos Clusters**:
    - Para cada ponto eie_i, é calculada a distância entre ele e cada protótipo de cluster. O ponto é atribuído ao cluster cujo protótipo está mais próximo.

- A distância é calculada como a soma das diferenças quadradas entre as características do ponto e do protótipo:

```
for (int i = 0; i < n; ++i) {
    double min_dist = std::numeric_limits<double>::max(); // Inicializa a
distância com um valor muito alto.
    int best_cluster = 0; // Atribui inicialmente ao primeiro cluster.

    // Calcula a distância de cada ponto a cada protótipo.
    for (int j = 0; j < k; ++j) {
        double dist = 0.0;
        for (size_t m = 0; m < view1[i].size(); ++m) {
            dist += (view1[i][m] - prototypes[j][m]) * (view1[i][m] -
prototypes[j][m]);
        }
        if (dist < min_dist) {
            min_dist = dist;
            best_cluster = j;
        }
    }
    clusters[i] = best_cluster; // Atribui o ponto ao cluster com a menor
distância.
}
```

4. **Passo 2 - Atualização dos Protótipos**:
    - Para cada cluster, o protótipo é recalculado. O novo protótipo é a média dos pontos que pertencem a esse cluster. Ou seja, a média das coordenadas dos pontos do cluster.

```
for (int j = 0; j < k; ++j) {
    std::vector<double> new_prototype(view1[0].size(), 0.0);
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (clusters[i] == j) {
            for (size_t m = 0; m < view1[i].size(); ++m) {
                new_prototype[m] += view1[i][m];
            }
            count++;
        }
    }

    // Calcula o novo protótipo
    if (count > 0) {
        for (size_t m = 0; m < new_prototype.size(); ++m) {
            prototypes[j][m] = new_prototype[m] / count;
        }
    }
}
```

5. **Passo 3 - Exibição dos Resultados**:
    - Após cada iteração, o código exibe a atribuição de cada ponto ao seu respectivo cluster, indicando ao usuário como a segmentação dos dados evolui ao longo das iterações.

```
std::cout << "Iteração " << iter + 1 << ":\n";
for (int i = 0; i < n; ++i) {
    std::cout << "Ponto " << i + 1 << " pertence ao cluster " << clusters[i] + 1
<< "\n";
}
```

6. **Determinação do Protótipo com Maior Distância**:

- Após 10 iterações, o algoritmo utiliza a função `std::max_element` para encontrar o protótipo com a maior distância, o que pode indicar um ponto de grande variação entre os clusters.

```
auto maxElement = std::max_element(prototypes.begin(), prototypes.end(),
                                   [](const std::vector<double>& a, const
std::vector<double>& b) {
                                       double sum_a = 0, sum_b = 0;
                                       for (size_t i = 0; i < a.size(); ++i) {
                                           sum_a += a[i] * a[i];
                                           sum_b += b[i] * b[i];
                                       }
                                       return sum_a < sum_b;
                                   });

int maxIndex = std::distance(prototypes.begin(), maxElement);
std::cout << "O protótipo com maior distância é o protótipo " << maxIndex + 1 <<
"\n";
```

Função `main`:

Na função principal, duas visões de dados são definidas, representadas por duas matrizes de características. A função `multiViewKMeans` é então chamada para realizar o agrupamento. Cada visão (ou matriz de dados) é usada para calcular as distâncias entre os pontos e os protótipos, ajudando a realizar o agrupamento de forma eficiente.

```
int main() {
    // Exemplo de duas visões de dados
    std::vector<std::vector<double>> view1 = {
        {1.0, 2.0},
        {1.5, 1.8},
        {5.0, 8.0},
        {8.0, 8.0},
        {1.0, 0.6}
    };

    std::vector<std::vector<double>> view2 = {
        {0.0, 1.0},
        {0.1, 0.9},
        {1.0, 0.0},
        {0.9, 0.1},
        {1.5, 2.0}
    };

    int k = 2; // Número de clusters

    // Chamada da função de k-means multi-visão
    multiViewKMeans(view1, view2, k);

    return 0;
}
```

Resultados do código:

Iteração 1:
Ponto 1 pertence ao cluster 1

Ponto 2 pertence ao cluster 1
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 2:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 3:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 4:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 5:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 6:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 7:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 8:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
Iteração 9:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2

Iteração 10:
Ponto 1 pertence ao cluster 2
Ponto 2 pertence ao cluster 2
Ponto 3 pertence ao cluster 1
Ponto 4 pertence ao cluster 1
Ponto 5 pertence ao cluster 2
O protótipo com maior distância é o protótipo 1