Trabalho Final de Inteligência Artificial: Estudo de métodos para Problemas de Agrupamento

Gabriel Ferrari Cipriano [1]

Ufes — Departamento de Informática [2]

Abstract

O agrupamento de dados de acordo com suas características é um dos desafios mais recorrentes nas aplicações de Aprendizado de Máquina. Este artigo busca realizar uma comparação experimental entre diferentes técnicas de agrupamento. Lidando com *Observações* numéricas e multidimensionais, três metaheurísticas serão treinadas e testadas, são elas: *GRASP, Simulated Annealing* e *Genetic Algorithm*.

Keywords: metaheuristics, clustering, Simulated Annealing, GRASP, Genetic Algorithm

1. Introdução

O objetivo deste estudo é realizar uma comparação de diferentes técnicas de agrupamento por meio de uma análise experimental. As metaheurísticas implementadas para o problema de agrupamento foram GRASP (*Greedy Randomized Adaptive Search Procedure*), *Simulated Annealing* e *Genetic Algorithm*, onde terão seus hiperparâmetros ajustados na etapa de treino, e posteriormente terão seus desempenhos avaliados de maneira comparativa com o método *KMeans*[3] da biblioteca *Scikit-learn*[1].

[1] gabriel.cipriano 'at' edu.ufes.br

[2] Ufes - Dep. de Informática https://informatica.ufes.br/

[3] sklearn.cluster.KMeans https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

Na seção 2 descreveremos o problema *K-Médias* e suas definições. Na seção 3 descreveremos os métodos implementados e suas particularidades. Na seção 4 apresentaremos os resultados das etapas de treino (4.1) e teste (4.2). Por fim, na seção 5 apresentaremos uma análise geral dos resultados, bem como contribuições e melhorias para trabalhos futuros.

2. O Problema K Médias

O problema de agrupamento de dados multidimensionais consiste em dividir os dados de entrada em um número determinado de grupos, se comportando como pontos em um espaço multidimensional. Quando um ponto pertence a um determinado grupo, significa que ele é mais relacionado com os pontos que estão naquele grupo, e menos relacionado com os pontos que não estão.

Existem diversos tipos de abordagens de *clusterização* na Análise de Agrupamento de Dados[2]. Neste estudo, o número de partições, também chamado de k, é previamente conhecido na hora de agrupar, sendo fornecido como entrada do problema. Os conjuntos das soluções serão Disjuntos, portanto, cada observação pertencerá a um único grupo, e o tipo de *cobertura* será Completa, isso é, todas as observações da entrada pertencerão a algum grupo.

Para que possamos avaliar a qualidade da solução de algum método de agrupamento, utilizaremos a soma das distâncias euclideanas quadradas (SSE) entre os pontos e o centro de massa de seu respectivo grupo. A fórmula é descrita abaixo:

$$\sum_{j=1}^{K} \sum_{x_i \in C_j} ||x_i - \mu_j||^2$$

Onde $||x_i - \mu_j||^2$ é a distância Euclideana quadrada entre o ponto x_i e o centróide μ_j centro de massa do grupo C_j.

O centróide μ de um grupo C de n pontos pode ser calculado da seguinte forma:

$$\mu = \frac{1}{n} \sum_{x_i \in C} x_i$$

No problema K-Médias o objetivo é distribuir os pontos em grupos de forma que minimize a SSE, se mostrando um problema de otimização. Ao comparar soluções diferentes, o conjunto de solução que obtiver o menor o valor SSE sinaliza que está mais próximo de uma solução ótima do que os outros.

3. Métodos Utilizados

As metaheurísticas escolhidas foram implementadas com base em pseudo-códigos genéricos apresentados em classe[2], tendo como desafio adaptá-los para o problema de clusterização de maneira minimamente satisfatória. Para realizar essa tarefa, foi utilizado a linguagem de programação *Python*[4] e bibliotecas auxiliares para manipulação e tratamento de Dados, como *Numpy*[5] e *Scipy Stats*[6], ferramentas que serão utilizadas também nas etapas seguintes deste trabalho.

3.1. Representação do Descritor de Espaço de Estados

Para auxiliar a explicação do Descritor de Espaço de Estados e dos Métodos, adotemos as seguintes *aliases*:

- K = Quantidade de grupos.
- N = Número de observações (pontos).
- D = Dimensão dos pontos.

Para lidar com uma instância de um problema de clusterização, foi criada a classe ***Clustering***, que armazena a lista bidimensional NxD que possui todos os N pontos de D dimensões a serem agrupados. A classe armazena também informações como o Número de pontos e a Dimensão deles, e fornece *métodos* que são comuns ao problema de clusterização, como a função *generate_initial_centroids* que gera K centroides iniciais baseado nos pontos de entrada.

[4] https://docs.python.org/3/
[5] https://numpy.org/doc/stable/reference/index.html
[6] https://docs.scipy.org/doc/scipy/reference/stats.html

Cada ponto do problema é representado como um *Numpy Array* imutável de tamanho D, onde cada item deste *array* é o valor do atributo de seu respectivo índice. O tipo *Numpy Array* (também chamado aqui de *array* ou lista) foi usado extensivamente neste trabalho por se mostrar muito mais eficiente computacionalmente que o tipo *List*, a lista padrão de python.

De fato, as três principais Estruturas de Dados do problema de clusterização foram implementados com *Numpy Arrays*, são elas:

- *Data*: Os dados de entrada. Uma lista NxD com todas as *observações*. O *index* de cada ponto representa sua ID.
- *Centróides*: Uma lista KxD que armazena os K centróides centros de massa dos K grupos. O *index* de cada centróide representa sua ID.
- *Labels*: Lista de tamanho N que guarda o ID do grupo atribuído de cada ponto. O *index* de cada *label* é também a ID do ponto correspondente.

Inspirado no *KMeans*, os estados de um problema podem estar representado em duas formas distintas: Como a lista de *labels*, cujo uma perturbação de estado possível é atribuir uma *label* diferente a algum ponto; E representado como a lista de centroides, que pode ser calculado a partir das *labels*, e vice-versa.

Com as principais estruturas de dados do problema sendo Listas e o acesso aos dados sendo direto (baseado em *indexes*), a manipulação e computação desses dados se tornou bastante efetiva. Utilizando conceitos de *List Comprehension* de *Python* e noções de álgebra linear apoiadas por funções do *Scipy* como a ***cdist***[7], que calcula a distância de cada ponto pra cada, é possível escrever um código conciso, claro e efetivo, embora tenha suas limitações de desempenho por *Python* ser uma linguagem interpretada.

Uma fase importante para o sucesso do agrupamento realizado pelo *KMeans* e outros algoritmos de agrupamento é a fase de disposição dos centróides iniciais, podendo ser abordada por diversas estratégias como o método *K-Means++*[3],

[7] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html

muito relevante na literatura de clusterização. O método de inicialização implementado para este estudo foi um dos mais simples conhecidos: Posicionar os centróides em K *observações* aleatórias dos dados de entrada.

3.2. GRASP

Greedy Randomized Adaptive Search Procedure é uma metaheurística onde suas iterações são compostas por duas etapas: Uma gulosa aleatória e outra de busca local, onde ao final da iteração é comparado a soluçao da iteração atual com a melhor solução anterior, se adaptatando e mantendo a solução que apresentou o melhor resultado de acordo com a funçao objetivo[4].

Os hiperparâmetros do *GRASP* são o Número máximo de iterações e o Número de Melhores vizinhos. Além do número máximo de iterações, outro critério de parada adotado é o tempo máximo de execução (1 segundo por padrão).

A fase gulosa-aleatória do algoritmo desenvolvido consiste em inicializar os K centroides e realizar a construção gulosa inspirada na estratégia *Hill Climbing*[8], onde é gerada uma vizinhança de estados atribuindo *labels* diferentes a determinados *pontos*, e sorteando um estado entre os n *melhores* da vizinhança para avançar.

Na fase de busca local é realizado um procedimento baseado no *KMeans*, onde as labels são calculadas a partir dos *centroides*, e em seguida os centroides são recalculados a partir das *labels* geradas, até um tempo máximo ou até convergir a um mínimo local (potencial candidato a mínimo global).

3.3. Simulated Annealing

Simulated Annealing é uma metaheurística de busca local randomizada que se baseia em conceitos da Termodinâmica, fazendo uma metáfora ao processo térmico de recozimento (*annealing*)[9]. No algoritmo, dada uma temperatura inicial T, existe uma probabilidade calculada por uma fórmula p de aceitar um

[8] https://www.sciencedirect.com/topics/computer-science/hill-climbing
[9] https://pt.wikipedia.org/wiki/Simulated_annealing

estado vizinho pior que o estado atual, e a medida que a temperatura abaixa, a probabilidade de saltar para um estado pior que o estado atual é cada vez menor, fazendo a solução convergir para um mínimo local.

Os hiperparâmetros do *Simulated Annealing* são a Temperatura inicial (T_0), a Taxa de decaimento da temperatura (α) e o Número de iterações. Os critérios de parada do algoritmo implementado são:

- *Tempo máximo de execução* = 1 segundo por padrão.
- *Temperatura mínima* = 0.01 por padrão.

No algoritmo desenvolvido o ciclo externo externo executa até um dos critérios de parada serem verdade, nele a lista de clusters é inicializada, uma vizinhança é gerada e entra no ciclo interno. No ciclo interno, que executa *n iterações* vezes, os *n melhores* vizinhos são sorteados e caso seu estado seja aceito, passa a ser o melhor estado da iteração atual, gerando uma nova vizinhança. A vizinhança é gerada atribuindo uma *label* diferente a um ponto aleatório. O algoritmo tende a passar um pouco do tempo de execução pois o tempo só é checado ao fim de cada ciclo interno.

Considere $\boldsymbol{f}$ a funçao objetivo, a probabilidade $\boldsymbol{p}$ de um estado $\boldsymbol{s}$ atual saltar para um estado $\boldsymbol{s'}$ de maior energia é dada pela seguinte fórmula:

$$p = e^{-\frac{f(s)-f(s')}{t}}$$

3.4. Genetic Algorithm

Genetic Algorithm é uma metaheurística de busca baseada na teoria da evolução de Charles Darwin, onde tenta reproduzir o processo de seleção natural de espécies e populações tomando emprestado conceitos como *mutação* e *cruzamento*. No algoritmo implementado, as *labels* de um estado são como os genes da fita de um DNA, e os centróides de um estado são como cromossomos, demonstrando que uma modificação num centróide pode significar uma alteração significativa nas *labels* daquela solução.

Os hiperparâmetros do *Genetic Algorithm* são o Tamanho da população, a Taxa de crossover e a Taxa de mutação. Os critérios de parada do algoritmo desenvolvido são:

- *Tempo máximo de execução* = 1 segundo por padrão.
- *Número máximo de iterações* = 3000 por padrão.
- *Número máximo de iterações sem melhora* = 30 por padrão.

O algoritmo desenvolvido começa uma população com um único indivíduo (lista de centroides), sendo este considerado como o melhor indivíduo nesta primeira iteração. Na fase de *Seleção*, é realizado um sorteio ponderado com base nas chances de cada indivíduo sobreviver - Quanto menor o SSE, mais chances de sobreviver - selecionando apenas um indivíduo como melhor da geração. Para gerar a população, o algoritmo causa perturbações na lista de *labels* deste melhor indivíduo, calculando os centróides de cada novo indivíduo a partir da *label* modificada.

3.4.1. Cruzamento

De acordo com a *taxa de crossover*, existe a probabilidade cruzamentos ocorrerem. O algoritmo realiza o cruzamento em dois indivíduos aleatórios da população, fazendo eles trocarem até K centróides (cromossomos) entre si.

3.4.2. Mutação

Existe a probabilidade mutações ocorrerem de acordo com a *taxa de mutação*. A mutação ocorre em algum individuo aleatório da população, modificando um número aleatório de *labels (genes)* deste indivíduo, recalculando a lista de centróides dele após isso.

4. Resultados dos Experimentos

Os experimentos realizados foram divididos em duas etapas: Etapa de Treino (4.1) onde é realizado o ajuste dos hiperparâmetros dos métodos e a Etapa de

Teste (4.2) onde os desempenhos dos algoritmos são comparados e avaliados. Em ambas etapas foi realizada a normalização dos resultados das médias com o Z-score por meio da função ***zscore***[10] disponibilizada pela biblioteca ***Scipy.Stats***.

Em ambas etapas foram utilizados os *datasets* ***Iris*** e ***Wine***, com 150 observações de 4 dimensões e 178 observações de 13 dimensões, respectivamente[5]. Na etapa de Teste também foi utilizado a base de dados ***Ionosphere*** que possui 351 observações de 34 dimensões, numa tentativa de avaliar o desempenho dos algoritmos sem um possível *Overfitting* por não ter feito parte da etapa de treino das metaheurísticas.

Os experimentos foram conduzidos num computador *Dual-Core Intel® Core™* i5-7360U CPU @ 2.30GHz, 16GB de memória RAM 2133MHz LPDDR3, 260GB APPLE® SSD AP0256J, *Intel Iris Plus Graphics* 640 1,5GB e Sistema Operacional MacOS. Foi definido um tempo máximo de execução de 1 segundo para os algoritmos avaliados, portanto o desempenho dos agrupamentos realizados pelos algoritmos pode variar a depender das configurações do computador, como o poder de processamento.

4.1. Treino

Nesta etapa foi realizada uma Busca em Grade (*Grid Search*) para que possamos identificar a melhor configuração de hiperparâmetros e esta seja levada para a etapa de Teste onde a comparação é realizada entre os métodos. Cada configuração foi executada 10 vezes para cada problema (*dataset, k*) e sua média normalizada (z-score) e média de tempo de execução foram obtidos. A Busca em Grade foi realizada com os seguintes hiperparâmetros:

- GRASP
  - *Numero de Iterações*: $[20, 50, 100, 200, 350, 500]$
  - *Numero de Melhores Elementos*: $[5, 10, 15]$
- Simulated Annealing

[10] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.zscore.html

  - T_0: $[500, 100, 50]$
  - α: $[0.95, 0.85, 0.7]$
  - *Numero de Iterações*: $[350, 500]$

- Genetic Algorithm
  - *Tamanho da população*: $[10, 30, 50]$
  - *Taxa de Crossover*: $[0.75, 0.85, 0.95]$
  - *Taxa de mutação*: $[0.10, 0.20]$

Cada configuração de hiperparâmetros possível foi treinada nos *datasets* ***Iris*** e ***Wine***, tendo seu desempenho avaliado para diferentes valores de K:

- Base de dados *Iris*: $K \in [3, 7, 10, 13, 22]$
- Base de dados *Wine*: $K \in [2, 6, 9, 11, 33]$

Na tabela 1 é apresentado as cinco melhores configurações de hiperparâmetros de cada método por Z-score médio:

	H.Parâmetro	Tempo Médio	Rank Médio	Z-score Médio
GRASP	(500, 10)	1.00	5.95	-0.60
	(350, 15)	1.01	6.65	-0.56
	(500, 15)	1.01	7.15	-0.53
	(500, 5)	1.00	6.75	-0.44
	(200, 15)	1.01	7.65	-0.32
Simulated Annealing	(500, 0.85, 500)	1.15	6.10	-0.71
	(500, 0.85, 350)	1.10	7.00	-0.45
	(500, 0.70, 500)	1.16	7.20	-0.45
	(500, 0.95, 500)	1.16	7.10	-0.41
	(100, 0.95, 350)	1.10	8.20	-0.29
Genetic Algorithm	(50, 0.75, 0.2)	0.97	5.60	-0.83
	(30, 0.75, 0.2)	0.78	6.60	-0.40
	(50, 0.85, 0.2)	0.97	7.20	-0.39
	(10, 0.95, 0.1)	0.29	8.90	-0.29
	(30, 0.85, 0.1)	0.74	7.90	-0.11

Tabela 1: Cinco melhores hiperparâmetros de cada método por Z-score.

4.1.1. GRASP

A figura 1 mostra o boxplot dos resultados normalizados de cada configuração de hiperparâmetros do método *GRASP*, bem como o boxplot dos tempos médios. É possível nota que para o *Número de Iterações = 20* os resultados ficaram bastante dispersos, demonstrando que o algoritmo tende a performar melhor para números de iterações maiores.

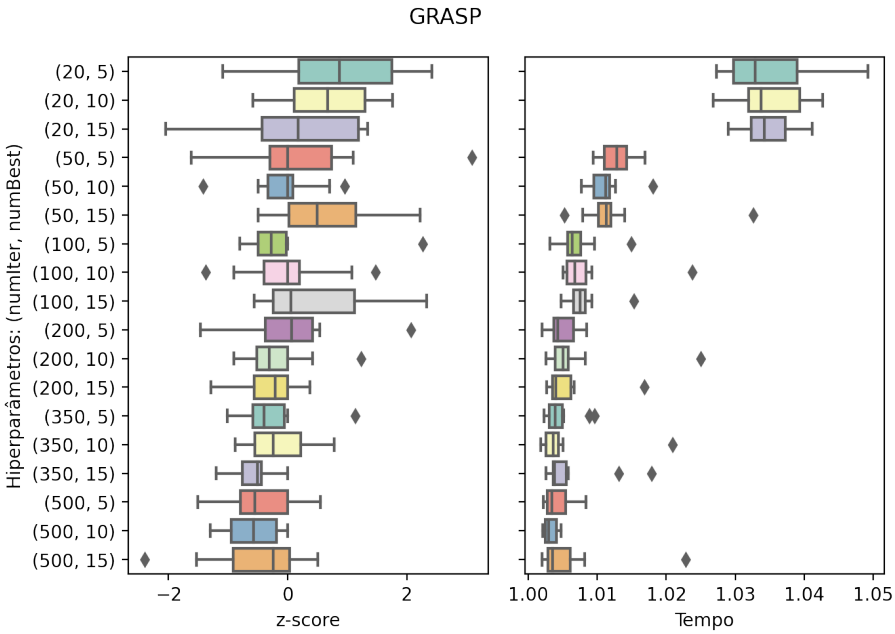

Figura 1: Boxplot - Z-score e tempo dos diferentes hiperparâmetros do método GRASP

Na tabela 2 é apresentado o ranqueamento médio de cada configuração de hiperparâmetros para cada problema *(dataset, k)* do algoritmo *GRASP*.

	Iris3	Iris7	Iris10	Iris13	Iris22	Wine2	Wine6	Wine9	Wine11	Wine33	**Média**
(500, 10)	9.5	1.0	6.0	3.0	9.0	9.5	7.5	4.0	2.0	8.0	**6.0**
(350, 15)	9.5	8.0	5.0	8.0	6.0	9.5	7.5	3.0	7.0	3.0	**6.7**
(500, 5)	9.5	3.0	11.0	6.0	3.0	9.5	7.5	12.0	5.0	1.0	**6.8**
(500, 15)	9.5	2.0	1.0	13.0	2.0	9.5	7.5	11.0	3.0	13.0	**7.2**
(200, 15)	9.5	5.0	8.0	10.0	11.0	9.5	7.5	9.0	1.0	6.0	**7.7**
(350, 5)	9.5	15.0	4.0	7.0	8.0	9.5	7.5	8.0	8.0	2.0	**7.8**
(350, 10)	9.5	4.0	14.0	11.0	4.0	9.5	7.5	6.0	13.0	5.0	**8.3**
(100, 5)	9.5	7.0	7.0	5.0	5.0	9.5	7.5	10.0	17.0	10.0	**8.8**
(200, 10)	9.5	10.0	3.0	12.0	16.0	9.5	7.5	7.0	4.0	9.0	**8.8**
(100, 10)	9.5	17.0	16.0	9.0	10.0	9.5	7.5	2.0	11.0	4.0	**9.6**
(50, 10)	9.5	11.0	9.0	15.0	7.0	9.5	7.5	1.0	12.0	16.0	**9.8**
(20, 15)	9.5	16.0	15.0	2.0	1.0	9.5	15.5	16.0	6.0	14.0	**10.4**
(200, 5)	9.5	6.0	2.0	14.0	18.0	9.5	7.5	14.0	15.0	12.0	**10.8**
(50, 5)	9.5	9.0	17.0	1.0	12.0	9.5	18.0	15.0	9.0	11.0	**11.1**
(100, 15)	9.5	18.0	18.0	17.0	13.0	9.5	7.5	13.0	10.0	7.0	**12.2**
(20, 10)	9.5	13.0	10.0	18.0	17.0	9.5	15.5	5.0	16.0	15.0	**12.8**
(50, 15)	9.5	12.0	13.0	16.0	14.0	9.5	7.5	18.0	14.0	17.0	**13.1**
(20, 5)	9.5	14.0	12.0	4.0	15.0	9.5	17.0	17.0	18.0	18.0	**13.4**

Tabela 2: GRASP - Ranqueamento dos hiperparâmetros em cada problema (*dataset, k*). Média de cada hiperparâmetro em destaque.

4.1.2. Simulated Annealing

A figura 2 mostra o boxplot dos resultados normalizados de cada configuração de hiperparâmetros do método *Simulated Annealing*, bem como o boxplot de seus tempos médios. Aqui os resultados se mostraram bastante dispersos e os tempos de execução muitas vezes passando pouco mais de 1 segundo, indicando a possibilidade de o tempo máximo de execução não ter permitido a Temperatura decrescer satisfatoriamente para diminuir a entropia.

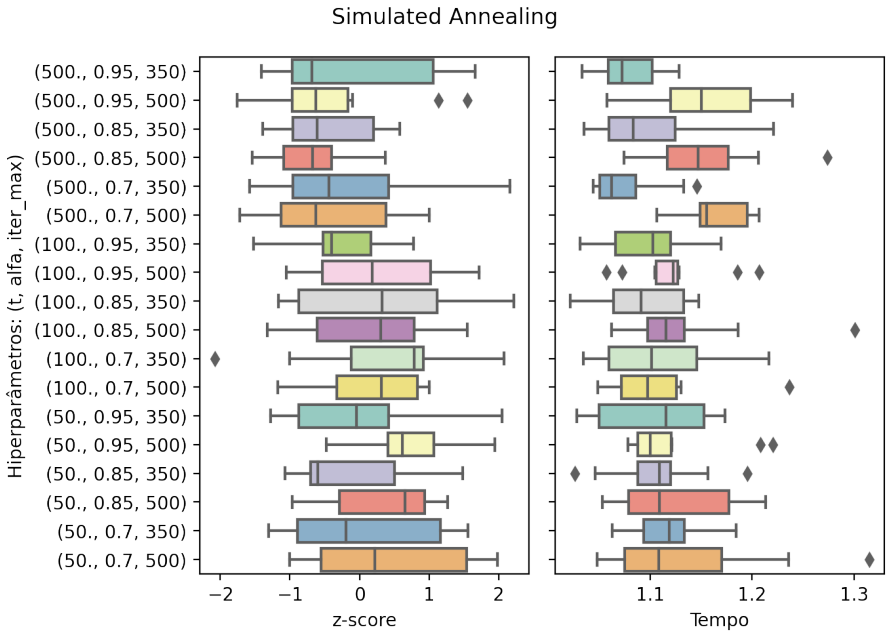

Figura 2: Boxplot - Z-score e tempo dos diferentes hiperparâmetros do método *Simulated Annealing*

Na tabela 3 é apresentado o ranqueamento médio de cada configuração de hiperparâmetros para cada problema *(dataset, k)* do algoritmo *Simulated Annealing*.

	Iris3	Iris7	Iris10	Iris13	Iris22	Wine2	Wine6	Wine9	Wine11	Wine33	**Média**
(500, 0.85, 500)	5.0	13.0	4.0	6.0	3.0	4.0	9.0	8.0	8.0	1.0	**6.1**
(500, 0.85, 350)	5.0	12.0	3.0	3.0	12.0	3.0	7.0	10.0	5.0	10.0	**7.0**
(500, 0.95, 500)	5.0	1.0	17.0	5.0	10.0	1.0	6.0	6.0	15.0	5.0	**7.1**
(500, 0.7, 500)	14.0	4.0	10.0	14.0	1.0	5.0	1.0	7.0	13.0	3.0	**7.2**
(100, 0.95, 350)	5.0	9.0	9.0	8.0	8.0	9.0	14.0	11.0	7.0	2.0	**8.2**
(500, 0.95, 350)	5.0	17.0	2.0	4.0	5.0	2.0	8.0	17.0	17.0	7.0	**8.4**
(50, 0.85, 350)	5.0	6.0	5.0	17.0	14.0	8.0	3.0	16.0	6.0	6.0	**8.6**
(500, 0.7, 350)	14.0	18.0	8.0	12.0	6.0	6.0	13.0	2.0	1.0	9.0	**8.9**
(50, 0.7, 350)	14.0	5.0	14.0	1.0	17.0	10.0	5.0	4.0	4.0	16.0	**9.0**
(50, 0.95, 350)	5.0	7.0	18.0	2.0	4.0	12.0	12.0	9.0	16.0	11.0	**9.6**
(100, 0.85, 500)	14.0	3.0	16.0	10.0	2.0	13.0	11.0	15.0	2.0	13.0	**9.9**
(100, 0.85, 350)	14.0	11.0	1.0	15.0	18.0	14.0	2.0	18.0	3.0	8.0	**10.4**
(100, 0.95, 500)	14.0	2.0	15.0	7.0	13.0	7.0	10.0	5.0	18.0	15.0	**10.6**
(100, 0.7, 500)	14.0	14.0	13.0	11.0	7.0	15.0	15.0	3.0	10.0	12.0	**11.4**
(50, 0.7, 500)	5.0	16.0	6.0	18.0	9.0	17.0	17.0	14.0	9.0	4.0	**11.5**
(50, 0.85, 500)	14.0	10.0	7.0	9.0	15.0	16.0	4.0	13.0	14.0	17.0	**11.9**
(100, 0.7, 350)	5.0	15.0	12.0	13.0	16.0	18.0	18.0	1.0	12.0	14.0	**12.4**
(50, 0.95, 500)	14.0	8.0	11.0	16.0	11.0	11.0	16.0	12.0	11.0	18.0	**12.8**

Tabela 3: Simulated Annealing - Ranqueamento dos hiperparâmetros em cada problema (*dataset, K*). Média de cada hiperparâmetro em destaque.

4.1.3. Genetic Algorithm

A figura 3 mostra o boxplot dos resultados normalizados de cada configuração de hiperparâmetros do método *Genetic Algorithm*, bem como o boxplot de seus tempos médios. É possível notar que para menores *tamanhos de população* o tempo de execução foi menor que tempo máximo, indicando que quando o tamanho da população é pequeno o algoritmo rapidamente converge para um resultado por conta da pouca variação genética numa população menor e graças ao critério de parada *"Número máximo de iterações sem melhora"*.

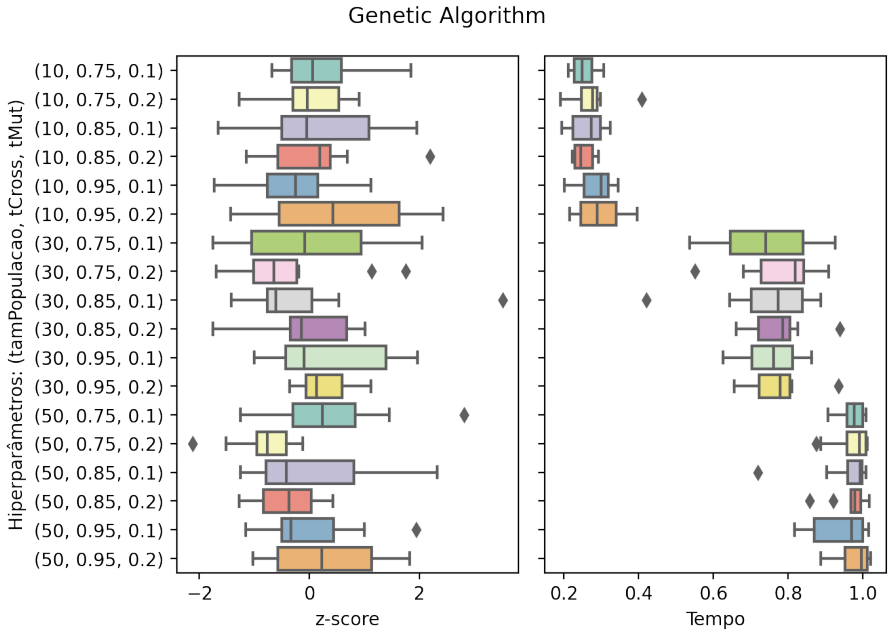

Figura 3: Boxplot - Z-score e tempo dos diferentes hiperparâmetros do método *Genetic Algorithm*

Na tabela 4 é apresentado o ranqueamento médio de cada configuração de hiperparâmetros para cada problema do algoritmo *Genetic Algorithm*.

	Iris3	Iris7	Iris10	Iris13	Iris22	Wine2	Wine6	Wine9	Wine11	Wine33	**Média**
(50, 0.75, 0.2)	10.0	1.0	8.0	4.0	5.0	5.0	8.0	1.0	8.0	6.0	**5.6**
(30, 0.75, 0.2)	5.0	2.0	4.0	1.0	3.0	8.0	7.0	16.0	17.0	3.0	**6.6**
(50, 0.85, 0.2)	2.0	10.0	3.0	3.0	7.0	10.0	16.0	2.0	11.0	8.0	**7.2**
(30, 0.85, 0.1)	18.0	13.0	7.0	6.0	1.0	9.0	4.0	4.0	5.0	12.0	**7.9**
(50, 0.85, 0.1)	4.0	16.0	15.0	5.0	2.0	1.5	11.0	18.0	6.0	4.0	**8.2**
(30, 0.75, 0.1)	1.0	18.0	16.0	2.0	11.0	5.0	1.0	13.0	18.0	2.0	**8.7**
(10, 0.95, 0.1)	12.0	3.0	6.0	12.0	14.0	16.0	5.0	10.0	1.0	10.0	**8.9**
(50, 0.95, 0.1)	17.0	4.0	14.0	9.0	12.0	5.0	13.0	7.0	7.0	7.0	**9.5**
(30, 0.85, 0.2)	6.0	8.0	5.0	7.0	9.0	15.0	14.0	15.0	15.0	1.0	**9.5**
(10, 0.75, 0.2)	13.0	15.0	9.0	15.0	4.0	13.0	2.0	9.0	12.0	9.0	**10.1**
(50, 0.95, 0.2)	3.0	9.0	2.0	17.0	10.0	7.0	15.0	6.0	16.0	17.0	**10.2**
(10, 0.85, 0.2)	9.0	12.0	13.0	13.0	8.0	18.0	6.0	3.0	13.0	13.0	**10.8**
(50, 0.75, 0.1)	8.0	11.0	12.0	14.0	16.0	1.5	18.0	11.0	3.0	14.0	**10.8**
(10, 0.75, 0.1)	11.0	6.0	17.0	16.0	13.0	12.0	12.0	8.0	9.0	5.0	**10.9**
(10, 0.85, 0.1)	15.0	5.0	11.0	11.0	18.0	17.0	3.0	14.0	2.0	16.0	**11.2**
(30, 0.95, 0.2)	7.0	14.0	10.0	10.0	15.0	11.0	10.0	12.0	10.0	15.0	**11.4**
(30, 0.95, 0.1)	16.0	7.0	18.0	8.0	6.0	3.0	9.0	17.0	14.0	18.0	**11.6**
(10, 0.95, 0.2)	14.0	17.0	1.0	18.0	17.0	14.0	17.0	5.0	4.0	11.0	**11.8**

Tabela 4: *Genetic Algorithm* - Ranqueamento dos hiperparâmetros em cada problema (*dataset, K*). Média de cada hiperparâmetro em destaque.

4.1.4. Resultados Alcançados

O SSE é uma função cujo estamos buscando uma minimização, para isso deve-se selecionar as configurações de hiperparâmetros que apresentaram os menores z-scores médios, bem como as que obtiveram os melhores ranqueamentos.

Para todos os algoritmos do nosso experimento a configuração que obteve o melhor ranqueamento médio também obteve o menor z-score médio (tabela 1), como mostra a tabela 5.

	Simulated Annealing	GRASP	Genetic Algorithm
Melhor Rank	(500, 0.85, 500)	(500, 10)	(50, 0.75, 0.2)
Melhor Z-score	(500, 0.85, 500)	(500, 10)	(50, 0.75, 0.2)

Tabela 5: Melhores hiperparâmetros de cada método por Rank e Z-score.

4.2. Teste

Nesta etapa do estudo buscamos avaliar as metaheurísticas comparando seus desempenhos juntamente ao método *KMeans* da biblioteca *Sci-kit learn*. Espera-se que o *KMeans* se destaque perante os demais pois é um algoritmo já consolidado e bastante otimizado. Cada algoritmo será executado 20 vezes para cada problema (*dataset, k*) e suas médias serão obtidas. Posteriormente, seus SSEs médios serão padronizados com o *z-score*.

Cada algoritmo foi executado nos *datasets* ***Iris***, ***Wine*** e ***Ionosphere***, tendo seus desempenhos avaliados para diferentes valores de K. É importante notar que o conjunto de valores de K na etapa de treino são diferentes dos valores de K da etapa de teste, numa tentativa de evitar *Overfitting*.

- Base de dados *Iris*: $K \in [2, 4, 8, 11, 15, 17, 23, 28, 32, 50]$
- Base de dados *Wine*: $K \in [3, 5, 13, 15, 20, 23, 25, 30, 41, 45]$
- Base de dados *Ionosphere*: $K \in [2, 3, 5, 10, 15, 20, 25, 30, 40, 50]$

Como visto na tabela 5, estes são os hiperparâmetros selecionados para a execução dos testes de cada algoritmo:

- GRASP
  - *Numero de Iterações* = 500
  - *Numero de Melhores Elementos* = 10
- Simulated Annealing
  - $T_0 = 500$
  - $\alpha = 0.85$
  - *Numero de Iterações* = 500
- Genetic Algorithm
  - *Tamanho da população* = 50
  - *Taxa de Crossover* = 0.75
  - *Taxa de mutação* = 0.20

Na tabela 6 é apresentado dados estatísticos da etapa de teste para cada método. Como esperado, é notório a diferença de desempenho do *KMeans* para os demais, apresentando os menores valores de z-score, desvio padrão e tempo de execução.

	Z-Score		Tempo	
	Média	Desv. Padrão	Média	Desv. Padrão
Simulated Annealing	0.07	0.77	1.31	0.26
GRASP	-0.53	0.41	1.01	0.01
Genetic Algorithm	1.33	0.53	1.02	0.03
KMeans	-0.87	0.36	0.15	0.12

Tabela 6: Dados estatísticos dos métodos testados.

A figura 4 apresenta o *boxplot* dos z-scores de cada método. Das metaheurísticas implementadas, *GRASP* foi a que apresentou o desempenho mais próximo do *KMeans*. A figura 5 apresenta o *boxplot* dos tempos de execução de cada método.

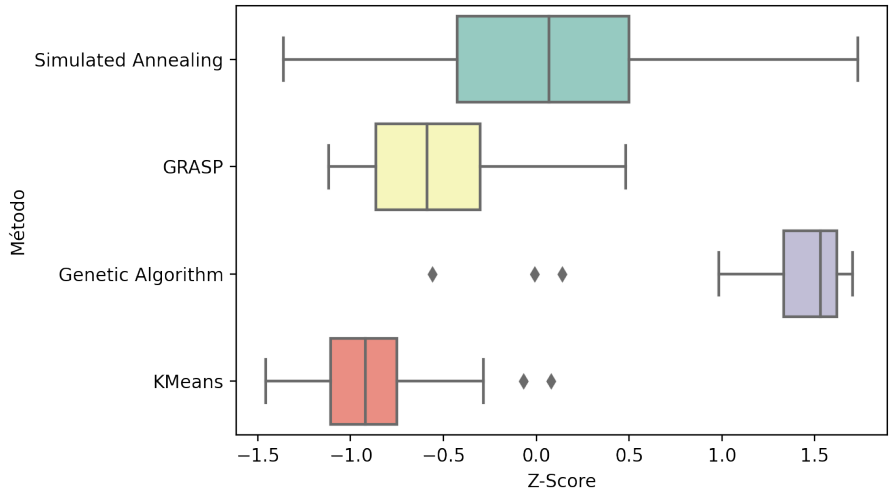

Figura 4: Boxplot das médias normalizadas de cada método.

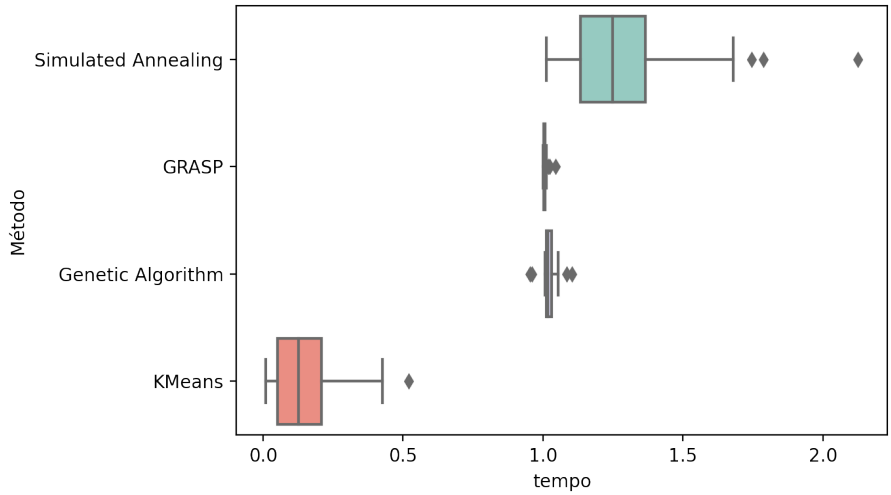

Figura 5: Boxplot dos tempos de execução de cada método.

A tabela 7 mostra o ranqueamento dos métodos para cada problema. É possível ver que o *Genetic Algorithm* implementado obteve o pior ranqueamento para a maior parte dos problemas, com rank médio de 3.87. Em contrapartida, o método *KMeans* teve o melhor ranqueamento médio, com 1.50.

	Sim. Annealing	GRASP	Genetic Alg.	KMeans
Iris2	4.00	2.00	3.00	1.00
Iris4	3.00	1.00	4.00	2.00
Iris8	3.00	1.00	4.00	2.00
Iris11	3.00	1.00	4.00	2.00
Iris15	2.00	3.00	4.00	1.00
Iris17	3.00	2.00	4.00	1.00
Iris23	3.00	2.00	4.00	1.00
Iris28	3.00	2.00	4.00	1.00
Iris32	3.00	2.00	4.00	1.00
Iris50	3.00	2.00	4.00	1.00
Wine3	4.00	1.50	3.00	1.50
Wine5	4.00	1.00	3.00	2.00
Wine13	3.00	2.00	4.00	1.00
Wine15	3.00	2.00	4.00	1.00
Wine20	3.00	2.00	4.00	1.00
Wine23	3.00	2.00	4.00	1.00
Wine25	3.00	2.00	4.00	1.00
Wine30	3.00	2.00	4.00	1.00
Wine41	2.00	3.00	4.00	1.00
Wine45	2.00	3.00	4.00	1.00
Ionos2	4.00	1.50	3.00	1.50
Ionos3	3.00	1.00	4.00	2.00
Ionos5	3.00	1.00	4.00	2.00
Ionos10	2.00	1.00	4.00	3.00
Ionos15	1.00	2.00	4.00	3.00
Ionos20	1.00	2.00	4.00	3.00
Ionos25	1.00	3.00	4.00	2.00
Ionos30	1.00	3.00	4.00	2.00
Ionos40	2.00	3.00	4.00	1.00
Ionos50	2.00	3.00	4.00	1.00
Rank Médio	**2.67**	**1.97**	**3.87**	**1.50**

Tabela 7: Ranqueamentos dos métodos para cada problema (*Dataset, k*). Ranqueamento médio em destaque.

A tabela 8 apresenta o melhor método geral e o melhor método dentre as metaheurísticas implementadas para este trabalho.

	Z-score Médio	Tempo médio	Rank Médio
KMeans	**-0.873**	**0.150**	**1.500**
GRASP	-0.532	1.008	1.967

Tabela 8: Melhores métodos. Melhor método geral (*KMeans*) em destaque e melhor método dentre os implementados (*GRASP*).

Foram realizados testes pareados com o Teste t de *Student* e o Teste de *Wilcoxon* para identificar se houveram diferenças estatísticas significativas dentre os resultados de cada método. A tabela 9 mostra o resultado destes testes.

Sim. Annealing	**0.00443**	**0.00001**	**0.00001**
0.00984	GRASP	**0.00000**	**0.00517**
0.00028	**0.00000**	Genetic Alg.	**0.00000**
0.00011	**0.00628**	**0.00000**	KMeans

Tabela 9: Teste pareado dos métodos. Teste t de *Student* na matriz triangular superior, Teste de *Wilcoxon* na matriz triangular inferior. Valores rejeitados (p-value $\leq$ 0,05) em destaque.

5. Conclusões

5.1. Análise Geral dos Resultados

Como apresentado, o método *Kmeans* foi o que obteve o melhor desempenho neste trabalho, o que já era esperado por ser um método simples e efetivo para o problema de *Clusterização*, embora bastante sujeito a mínimos locais. Outro fato que contribuiu para seu sucesso perante os demais foi sua implementação bastante eficiente, executando boa parte do processamento de dados de maneira compilada, vetorizada ou em *threads*.

O teste pareado confirmou que a diferença do *KMeans* para os demais foi realmente significativa. De fato, em todos os pares as diferenças foram significativas, nos permitindo afirmar que o *Kmeans* foi o melhor método, o *GRASP* foi o segundo melhor, e o *Genetic Algorithm* se mostrou o pior desempenho, dentro das limitações deste trabalho.

Foi possível notar que o principal critério de parada para os algoritmos desenvolvidos foi o Tempo de Execução. É possível citar dois fatores que contribuíram para isso, (1) A falta de otimização que carece de um conhecimento avançado da linguagem *Python*, (2) As metaheurísticas apresentadas são diferentes abordagens para a tentativa de escapar de mínimos locais e alcançar o mínimo global, por conta disso, elas vão demandar naturalmente maior tempo de execução pois não são tão simples como o método *Kmeans*.

5.2. Contribuições do Trabalho

Baseado nas metaheurísticas propostas foram desenvolvidos três algoritmos para o problema de agrupamento, que podem ser reutilizados e avaliados em outros *datasets* e conjuntos de problemas. Foi implementado também a classe ***Clustering***, que fornece métodos úteis que podem ser utilizados por outros algoritmos para o problema de clusterização.

5.3. Melhorias e Trabalhos Futuros

Como dito anteriormente, o limite de tempo de execução para os treinos e testes possivelmente impediu os algoritmos desenvolvidos de apresentarem resultados melhores. Um trabalho futuro poderia ser aumentar o limite de tempo de execução e avaliar se houveram diferenças significativas nos resultados.

É possível melhorar o desempenho do código implementado, o que não foi incentivado devido a natureza desse trabalho. Uma continuação deste trabalho poderia ser otimizar os algoritmos desenvolvidos, utilizando por exemplo *Cython*[11] para gerar código compilado, e repetir este estudo para conferir se alcança diferentes resultados.

É recomendado que trabalhos futuros tenham a normalização dos dados de entrada incentivada. Por se tratar de uma abordagem onde os dados são vistos como pontos num espaço multidimensional, a diferença de magnitude das dimensões pode causar distorções nos resultados, pois valores muito altos terão um peso muito maior no resultado do cálculo da SSE.

[11] https://cython.org/

6. Referências

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[2] F. M. Varejao, Curso de inteligência artificial - ufes, notas de aula (2021).

[3] D. Arthur, S. Vassilvitskii, K-means++: The advantages of careful seeding, in: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '07, Society for Industrial and Applied Mathematics, USA, 2007, p. 1027–1035.

[4] M. G. C. Resende, C. C. Ribeiro, Greedy randomized adaptive search procedures: Advances and extensions, Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol 272. (2019).

[5] D. Dua, C. Graff, UCI machine learning repository (2017).
URL http://archive.ics.uci.edu/ml