# Bisecting K-means

Inteligência na Web e Big Data
Prof. Dr. Fabrício Olivetti de França

Rafael Jeferson Pezzuto Damaceno - RA 141610015

## 1. Bisecting K-means

*Bisecting K-means* é um algoritmo para agrupamento de dados baseado no *K-means*. Algoritmos para agrupamento de dados são aqueles responsáveis por dividir um conjunto de dados em grupos ou para sumarização de dados. Há diversos tipos e características de algoritmos para agrupamento de dados. Os mais importantes são hierárquico, particional, exclusivo, *overlapping*, *fuzzy*, completo e parcial [1].

O agrupamento particional é aquele no qual os dados são divididos em grupos distintos. Hierárquico é aquele no qual os grupos possuem uma relação de hierarquia entre si, formando uma estrutura de árvore. Um agrupamento é exclusivo quando os objetos que formam os grupos não se repetem entre eles, ou seja, um dado objeto pertence a somente um grupo. De forma oposta, *Overlapping* é a situação em que um objeto pode pertencer a mais de um grupo. No agrupamento que usa *Fuzzy*, cada objeto contém uma probabilidade de pertencimento a cada um dos *clusters* considerados. No agrupamento completo todo objeto é inserido em um *cluster* e alguns objetos podem permanecer sem *cluster* definido [1].

É possível adotar diversas características das mencionadas para desenvolver um algoritmo de agrupamento. No caso do algoritmo *Bisecting K-means*, costuma-se adotar o agrupamento completo, hierárquico ou não, exclusivo. Os pseudos-códigos dos algoritmos *Bisecting K-means* e *K-means*, conforme propostos por Tan et al. [1], são mostrados a seguir:

```
BISECTING-K-MEANS(k, numeroTentativas)
1   Inicialize lista de clusters com todos os pontos
2   repeat
3       Remova um cluster da lista de clusters
4       for i = 1 até numeroTentativas do
5           Bifurque o cluster selecionado usando K-MEANS
6       end for
7       Selecione os dois melhores clusters da bifurcação
7       Adicione os dois clusters à lista de clusters
7   until Até que a lista de clusters contenha k clusters
```

```
K-MEANS(k, numeroIteracoes)
1   Selecione k pontos como centróides
2   repeat
3       Forme k clusters inserindo em cada um os pontos mais próximos
4       Recalcule o centróide de cada cluster
5   until Até que os centróides não mudemo ou que se atinga numeroIteracoes
```

Antes de prosseguir com as implementações dos algoritmos, cabe mencionar a modalidade de programação que foi utilizada, a programação funcional. Trata-se de um paradigma cuja principal característica é o uso de funções matemáticas, isto é, elementos que recebem um valor como entrada e retornam outro como saída. Dessa forma, um valor y obtido como saída será sempre o mesmo para um valor x inserido como entrada (princípio de Pureza) [2].

Esta e outras características da programação funcional conferem à ferramenta *PySpark* a capacidade de tratar dados de forma paralela e distribuída. Assim, o uso de funções conhecidas como `Map` e `MapReduce` aceleram o processo de obtenção de resultados, associados ao conhecido *Resilient Distributed Dataset* (RDD), uma abstração de base de dados que é imutável e particionada em vários conjuntos que podem ser trabalhados de forma paralela [3].

### *1.1. Implementação*

Foi utilizado *PySpark* para implementar de forma paralela o algoritmo *Bisecting K-means*, desde o pré-processamento da base de dados até a base do *K-means*. A implementação utilizou características de programação funcional em conjunto com processamento distribuído, tais como as funções `Map` (linhas 17,18 do **bisectKmeans**) e `ReduceByKey` (linhas 4,7-9 do **kmeans**) do *PySpark*. Os códigos-fontes em *Python* do *Bisecting K-means* e do *K-means* são mostrados a seguir:

```
1  def bisectKmeans(data, k, bisects, iteractions):
2      finalClusters = [data]
3      while(len(finalClusters) != k):
4          clusterToSplit = finalClusters.pop()
5          sse = []
6          tmpClusters = []
7          for i in range(bisects):
8              clustersRDD, centroids = kmeans(clusterToSplit, 2, iteractions)
9              tmpClusters.append(clustersRDD.filter(lambda x:x[0] == 0))
10             tmpClusters.append(clustersRDD.filter(lambda x:x[0] == 1))
11             sse1 = getSSE(tmpClusters[-2], centroids[0])
12             sse2 = getSSE(tmpClusters[-1], centroids[1])
13             sse.append(sse1 + sse2)
14         minSseIndex = np.argmin(sse)
15         largerCluster, minorCluster = getClustersByCount(tmpClusters[minSseIndex*2]
16                                                          , tmpClusters[minSseIndex*2 + 1])
17         finalClusters.append(minorCluster.map(lambda x:(x[1], x[2])))
18         finalClusters.append(largerCluster.map(lambda x:(x[1], x[2])))
19     return finalClusters
```

```
1  def kmeans(data, k, iteractions):
2      centroids = [p[1] for p in chooseRandomPoints(data, k)]
3      for i in range(iteractions):
4          clustersRDD = data.map(lambda x:(np.argmin([euclidianDistance(x[1], c) for c
5                                                      in centroids]), x[0], x[1]))
6          newCentroids = (clustersRDD
7              .map(lambda x:(x[0], [x[2],1]))
8              .reduceByKey(lambda x,y:([(np.array(x[0]) + np.array(y[0])) ,(x[1] + y[1])]))
9              .map(lambda x:list(np.array(x[1][0])/(x[1][1]))))
```

```
10            ).collect()
11          centroidsHaveChanged = [areCentroidsDifferent(centroids[i], newCentroids[i])
12                                                          for i in range(len(centroids))]
13          if True not in centroidsHaveChanged:
14              return clustersRDD, centroids
15          else:
16              centroids = newCentroids
17      return clustersRDD, centroids
```

Note que o *PySpark* paraleliza o cálculo dos novos centróides no **kmeans** (linhas 6 à 10). Isso é feito pelo uso da função `map`, sobre a RDD, que transforma, de forma paralela, um conjunto de objetos em outro. No caso em específico, primeiro mapeia-se cada objeto do RDD para uma lista, composta pelo terceiro elemento (uma lista de métricas) do objeto original, seguido do número 1. Este último servirá como acumulador para calcular o número de pontos de cada *cluster* e assim calcular o ponto mediano. A função `reduceByKey` agrupa todos os objetos que pertencem a um mesmo *cluster* (isto é, a chave de cada objeto é o código do cluster a que pertence) e realiza a operação de soma nas listas de métricas e no acumuladador (linha 9). Por fim, a função `map` (linha 10) divide cada lista de métricas pelo acumulador.

## 2. Base de Dados

A base de dados escolhida para aplicar o algoritmo *Bisecting K-means* foi desenvolvida pelo grupo de Cientometria da UFABC[1] e está em uso pela Plataforma Acácia[2]. Trata-se de uma base de dados genealógicos, em forma de grafo, que reúne 1 111 544 acadêmicos e 1 208 399 relações formais de orientação nos níveis de Mestrado e Doutorado. Cada vértice representa um acadêmico e cada aresta representa uma relação formal de orientação entre dois acadêmicos.

Cada vértice contém um código identificador único, o nome completo do acadêmico, a área do conhecimento em que atua, sua maior titulação, o ano de obtenção de sua maior titulação, o nome da instituição em que trabalha e 3 métricas genealógicas, a saber, fecundidade, descendência e índice genealógico. Para o escopo deste projeto, foram calculadas mais 10 métricas genealógicas, para complementar as informações dos vértices, a saber, fecundidade inversa, descendência inversa, fertilidade e fertilidade inversa, primo e primo inverso, orientações e orientações inversa, gerações e gerações inversa. A descrição do significado destas métricas e de como foram calculadas não está no escopo deste projeto. Para maiores detalhes de como o banco de dados foi gerado e sobre como as métricas métricas fecundidade, descendência e suas versões inversas foram calculadas, vide [4]. Para maiores detalhes sobre o índice genealógico, vide [5]. As informações consideradas para a aplicação do *Bisecting K-means* fora o código identificador de cada vértice e as treze métricas genealógicas.

## 3. Experimento

O experimento consistiu em aplicar o algoritmo *Bisecting K-means* para identificar diferentes *clusters* contendo vértices, com base nas informações das métricas. Inicialmente, procedeu-se ao pre-processamento e normalização dos vértices, transformado cada um em uma tupla formada pelo

[1] `pesquisa.ufabc.edu.br/cientometria`, último acesso 10 Mai 2018.
[2] `plataforma-acacia.org`, último acesso em 10 Mai 2018.

identificador do vértice e uma lista de números reais representando os valores das métricas. Como cada métrica possuía valores muito amplos (fecundidade poderia ir de 0 a 421, e descendência de 0 a 18 679), optou-se por normalizar as métricas conforme a técnica de z-score, que consiste em subtrair cada valor pela sua média (de todos os pontos para aquele valor) e dividir o resultado pelo desvio-padrão (de todos os pontos para aquele valor) [6].

Para comparar a eficiência do algoritmo paralelizado com o algoritmo sem paralelismo, utilizou o ambiente do *PySpark* com um núcleo de processamento e com doze núcleos de processamento. Para cada modalidade (paralela e não paralela), o algoritmo foi executado três vezes, para gerar 2, 5 e 10 clusters. O tempo de execução foi cronometrado utilizando a extensão `autotime`[3] no próprio ambiente *PySpark*, configurado com Python 3.6.5. O computador utilizado para rodar os experimentos possui um processador AMD Ryzen 1600 (com 6 núcleos/12 *threads* e clock de 3.4 GHz em todos os núcleos), 16 GB de memória RAM (com frequência a 3200 MHz) e SDD Sandisk 240 GB (modelo G36).

## 4. Resultados

A Figura 1 ilustra os tempos de execução (eixo x) em segundos para cada modalidade (12 núcleos-paralela, 1 núcleo-não paralela), para obter 2, 5 e 10 *clusters*, considerando 1 111 544 vértices. A configuração do algoritmo foi $k = 2, 5, 10$, *bisects* $= 2$ e *iteractions* $= 5$. Os dados mostram que o uso de paralelismo, possibilitado pelas funções `Map` e `ReduceByKey` reduziram o tempo em grande parte. Para todos os casos, o tempo foi quase duas vezes menor para doze núcleos, quando comparado ao uso de somente um núcleo.

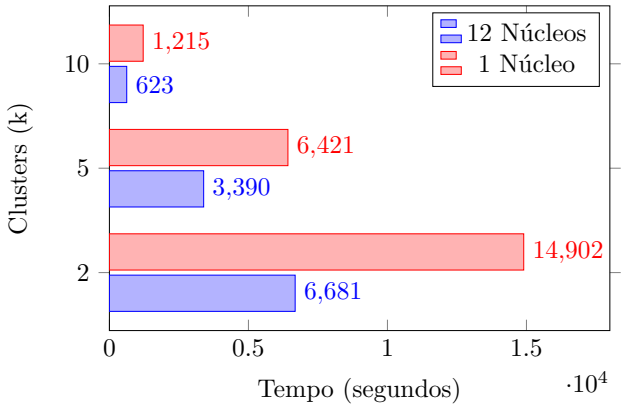


Figura 1: Tempo de execução (eixo x) do algoritmo rodando em paralelo (12 núcleos) e em não paralelo (1 núcleo) para identificar 2, 5 e 10 *clusters* no conjunto de 1 111 544 vértices.

Como trabalho futuro está a verificação da qualidade e do significado dos agrupamentos realizados, utilizando, para isso, informações extras dos vértices, tais como idade acadêmica (tempo em anos em que um acadêmico é Mestre ou Doutor) e área do conhecimento. Deseja-se investigar se as métricas topológicas possuem algum padrão conforme estas informações extras. Verificar se os vértices agrupados possuem algo em comum, tal como idade acadêmica ou as áreas do conhecimento é uma das metas. Ainda, verificar se alguma área do conhecimento esteve presente em maior intensidade em um dos *clusters* é outra meta.

[3] `github.com/cpcloud/ipython-autotime`, acesso em 10 Mai 2018.

## Referências


[1] P. N. Tan, et al., Introduction to data mining, Pearson Education India, 2006.

[2] S. Thompson, Haskell: the craft of functional programming, volume 2, Addison-Wesley, 2011.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets., HotCloud 10 (2010) 95.

[4] R. J. P. Damaceno, L. Rossi, J. P. Mena-Chalco, Identificaçao do grafo de genealogia acadêmica de pesquisadores: Uma abordagem baseada na plataforma lattes, in: Proceedings of the 32nd Brazilian Symposium on Databases, volume 1, pp. 76–87.

[5] L. Rossi, I. L. Freire, J. P. Mena-Chalco, Genealogical index: A metric to analyze advisor–advisee relationships, Journal of Informetrics 11 (2017) 564–582.

[6] L. Al-Shalabi, Z. Shaaban, B. Kasasbeh, Data mining: A preprocessing engine, Journal of Computer Science 2 (2006) 735–739.