Paralelização do k-Means com implementação em linguagem Python

Samuel Caetano da Silva

7 de dezembro de 2017

Resumo

O algoritmo k-Means é amplamente utilizado em aplicações de agrupamentos de dados. Essa técnica consiste em agrupar n dados amostrais em k grupos. Esse relatório propõe uma versão paralela dessa algoritmo, com o objetivo de diminir o tempo de agrupamento para grandes quantidades de dados. Ao final, é apresentado uma comparação de implementação em diferentes módulos da linguagem Python.

1 Introdução

A Mineração de Dados (MD) é uma das tecnologias mais capazes de possibilitar o nosso entendimento com relação ao mundo externo, pois consiste numa aplicação de técnicas matemáticas e estatísticas para reconhecer padrões, descobrir corelações e realizar predições dentro de um conjunto grande de dados (LAROSE, 2005).

A tarefa de agrupar dados é muito importante dentro do contexto atual, onde a quantidade de dados disponível é muito grande. Entretanto, esses dados são na maioria das vezes não formatados e não rotulados e isso quer dizer que os dados por si só não possibilitam a obtenção de conhecimento, fazendo-se necessário a aplicação de técnicas de MD que permitam que esses padrões sejam descobertos.

A clusterização por k-Means é relevante nesse contexto, onde a quantidade de dados é grande, mas a capacidade de processamento de máquina, nem tanto. Desse modo, a paralelização de algoritmos sequênciais se torna

uma tendência por ser capaz de possibilitar uma redução no tempo de processamento de uma tarefa.

2 O algoritmo k-Means sequencial

A técnica de clusterização pode ser compreendida em duas formas: hierárquica e particional. A clusterização hierárquica consiste numa construção de clusters semelhantes a árvores, chamado de *dendrograma*, através de métodos divisivos e aglomerativos. Já a clusterização particional visa particionar um conjunto com n dados entre grupos diferentes. A clusterização por k-means é particional (LAROSE, 2005).

Esse algoritmo consiste em verificar as distâncias entre as posições das amostras e as posições dos centros dos clusters, também chamada de centróide. Para cálculo dessas distâncias, nesse trabalho, utilizou-se a distância euclidiana entre os pontos, que consiste na equação (1):

$$dist_{euclidiana} = \sqrt{\sum_{i=1}^{m}(x_i - y_i)^2}, \tag{1}$$

, onde m é a dimensão de análise, ou seja $\mathbb{R}^m$, x e y são representações vetoriais dos dados, neste caso x pertence à coleção de amostras e y pertence à coleção dos centros dos clusters.

Para identificar se o agrupamento atual é bom, os métodos de clusterização verificam se a similaridade dentro do *cluster* é muito alta, quando comparada com a similaridade entre amostras de outros *clusters*. A figura 1 ilustra esse situação.

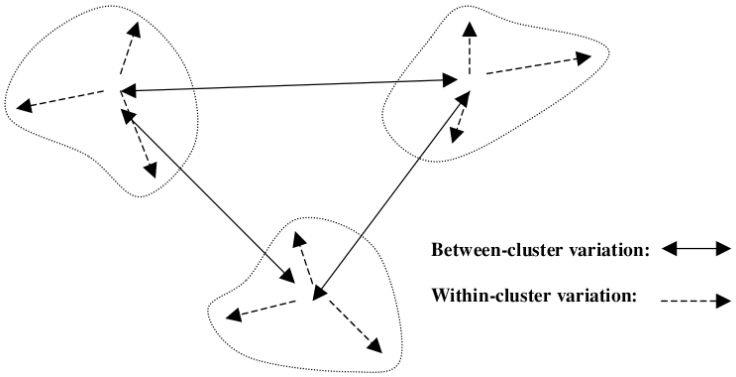

Figura 1: Similaridades dentro e fora dos clusters.

Para cálculo da variação interna ao *cluster* (*between cluster variation, bcv*) usou-se a equação (2) e para cálculo da variação externa ao cluster (*within cluster variation, wcv*), (3). O algoritmo do k-means é o algoritmo 1.

$$bcv = \sum_{i=1}^{m}(x_i - y_i)^2, \tag{2}$$

$$wcv = \sum_{i=1}^{k}(\sum_{p \in cluster[i]} distancia(x_i, y_i)^2), \tag{3}$$

.

Esse algoritmo possui complexidade que varia com a quantidade de dados e como a quantidade de dados normalmente é muito grande, dentro do contexto de MD, realizar iterações pode ser uma tarefa muito custosa e consequentemente pode tornar a tarefa de agrupammento um processo muito lento.

3 A proprosta do algoritmo paralelo

Observando-se o algoritmo serial do k-means, percebe-se que existem dois pontos críticos, que são duas situações de *loop*. A primeira, é o laço que acontece enquanto não houver convergência, essa convergência é atingida quando não há mais variação no mínimo local da função de convergência, significando que o vale ideal foi atingido; em outras palavras: quando não há

Algorithm 1 Algoritmo k-Means

```
1: procedure K-MEANS
2:     k ← número de clusters
3:     posições iniciais dos k clusters ← valor aleatório
4: top:
5:     for amostra da coleção de amostras do
6:         for posição da coleção de posições dos clusters do
7:             distancias ← distância euclidiana entre a amostra e a posição
8:         índice ← min(distancias)
9:         cluster[índice] ← amostra
10:    custo ← variação entre clusters/variação dentro do cluster
11:    for cluster na coleção de clusters do
12:        centroide ← calculo do centroide do cluster
13:        atualize a posicão do cluster, relativo ao centroide
14:    se custo difente do custo anterior goto top
```

mais variação entre o custo atual calculado e o custo imediatamente anterior. O segundo laço crítico é aquele que calcula e agrega a amostra ao grupo correto..

O problema do primeiro caso está que a convergência pode demorar muito para ser atingida, não sendo possível prever de antemão quantos passos seriam necessarios para atingí-la. Já no segundo, tem-se um laço $O(N)$ e em situações reais de MD temos que $N \to \infty$, no sentido que infinito está nas casas das dezenas ou centenas de milhares.

Em face desses problemas, deve-se paralelizar ambos, ou pelo menos um, desses laços críticos. A proposta apresentada nesse trabalho consistiu na paralelização do segundo laço crítico.

A figura 2 ilustra o esquema proposto, que consiste em realizar uma distribuição de cargas para t threads, de modo que o mapeamento do dado para seu respectivo grupo aconteça simultaneamente, visto que cada thread terá um segmento do conjunto de dados. As setas de duplo sentido na figura indicam propagação de ida e de volta.

Essas threads mapeadoras foram chamadas de *mappers*. Todos os *mappers* escrevem sua parte clusterizada num cluster global, de modo que uma thread atualizadora, chamada *updater*, realiza a atualização das posições dos centróides dos clusters de acordo com o conteúdo que está no cluster global,

que é a propagação de ida. Enquanto o *updater* não atingir convergência, ele pede para que os *mappers* mapeiem novamente as amostras com as posições atualizadas do cluster global, que é a propagação de volta. Quando a convergência é atingida, um *flag* global é acionado e isso sinaliza para os *mappers* que eles devem encerrar.

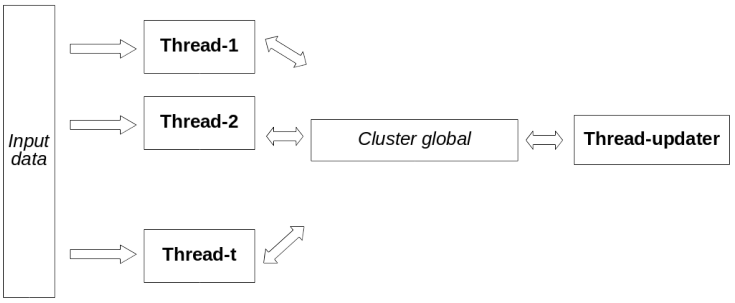

Figura 2: Diagrama do k-means paralelo.

4 Resultados da implementação em linguagem Python

Para verificar a validade dessa proposta escolheu-se implementá-la em alguma linguagem de programação popular. A escolha da linguagem Python deu-se pela popularidade dessa e pela eficiência de seu uso em aplicações de MD, MT, redes neurais e outros campos do *Machine Learning*.

Os testes foram realizados numa máquina com processador Intel Pentium, com 2.16GHz e 4 núcleos. A versão do Python implementada foi a 2.7. Os módulos de programação concorrente escolhidos foi o *threading* e o *multiprocessing*

Essa escolha deu-se pela diferença conceitual que há em ambas. O módulo *threading* realiza a construção de threads, enquanto que o módulo *multiprocessing* realiza o *spawing* de processos. A escolha do *threading* é recomendada para aplicações I/O Bound, enquanto que o *multiprocessing* é uma opção para aplicações CPU Bound[1].

[1]http://uwpce-pythoncert.github.io/SystemDevelopment/threading-multiprocessing.html

O uso de threads em Python não permite ganho significativo no uso dos diferentes núcleos de uma máquina devido ao Global Interpreter Lock (GIL), que é um mecanismo que faz com que apenas uma thread execute um byte-code por vez, permitindo uma espécie de concorrência e não paralelismo.

Em contra partida, o módulo *multiprocessing* consegue evitar o GIL por causa do uso de sub-processos no lugar de threads. Assim, esse módulo permite ao desenvolvedor utilizar os múltiplos processadores de uma máquina, permitindo uma espécie de paralelismo.

Os testes implementados foram verificados com 1000, 10000 e 100000 amostras, que são pares de números gerados aleatóriamente. Os *mappers* foram 2, 4 e 8.

As figuras de 3 à 6, mostram o tempo de execução para cada implementação.

Figura 3: Comparação de execução com 5 clusters numa amostra de 1000 elementos.

Figura 4: Comparação de execução com 5 clusters numa amostra de 10000 elementos.

Figura 5: Comparação de execução com 5 clusters numa amostra de 100000 elementos.

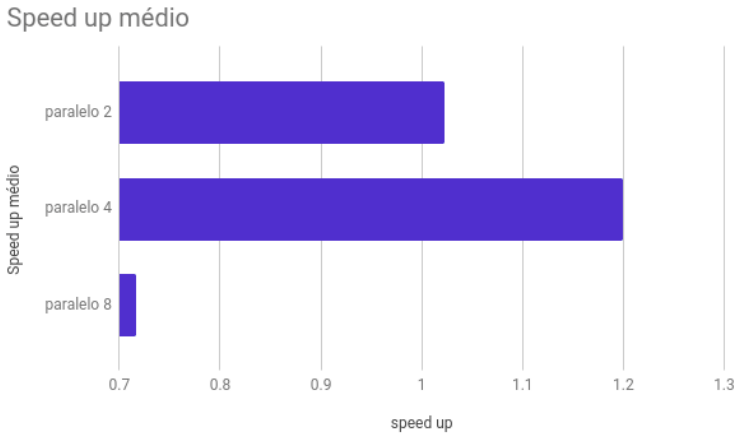

Figura 6: Speed up obtido entre a versão paralela e a sequencial.

As figuras 7, 8, e 9, mostram que a corretude foi mantida após a paralelização do k-means sequencial.

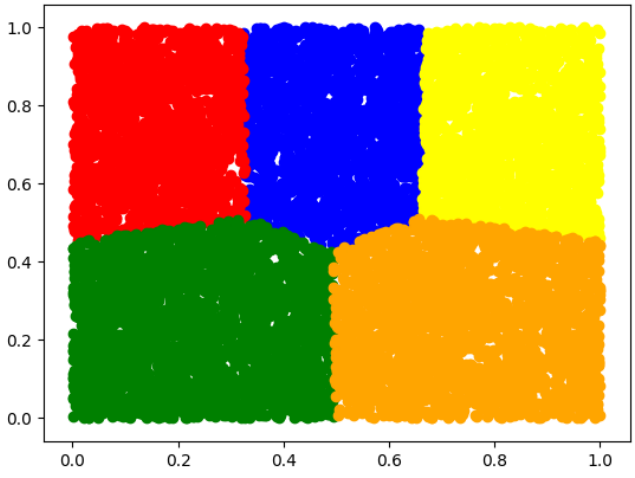

Figura 7: Clusters descobertos pela versão sequencial.

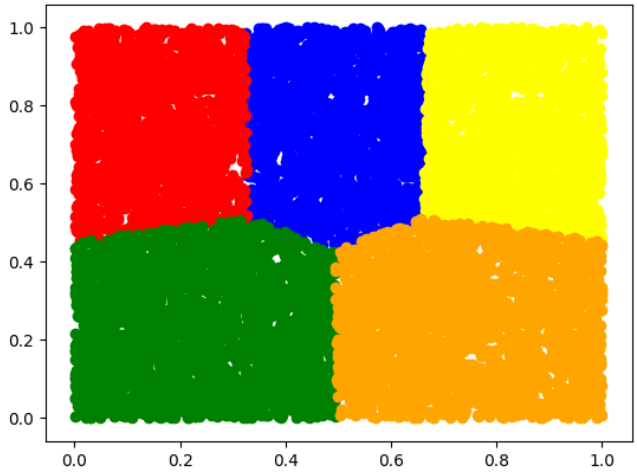

Figura 8: Clusters descobertos pela versão concorrente.

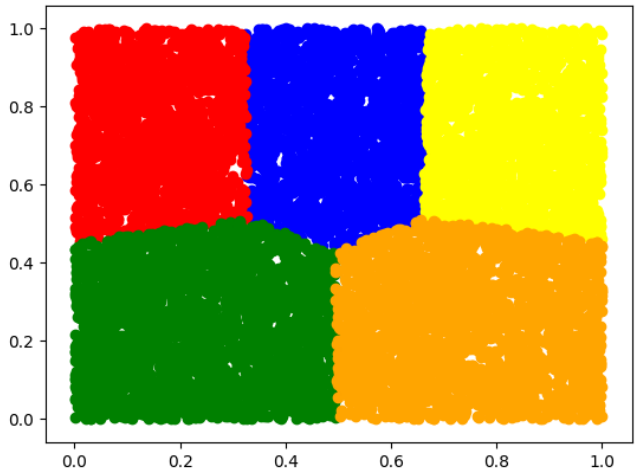

Figura 9: Clusters descobertos pela versão paralela.

5 Conclusão

Após realizar os testes concluiu-se que a implementação de um código paralelo em Python pode ser realizada através do módulo *multiprocessing*. O *speedup* conseguido foi de quase 2 vezes com relação ao tempo sequencial, com 4 *mappers*.

No geral, a linguagem Python não se mostra como opção ótima para paralelização de algoritmos, considerando o módulo *threading*, sendo necessária a utilização de *workarounds* para poder contornar o problema do GIL.

Entretanto, essa proposta de paralelização do k-means pode ser melhorada, na presente os *mappers* ficam ociosos durante o tempo de execução do *updater*, uma solução é paralelizaar também a tarefa de atualização dos centróides.

Referências

[1] Larose, D. T. *Discovering Knowledge in Data*. John Wiley & Sons, Inc., 2005.

[2] threading module, `https://docs.python.org/3/library/threading.html#module-threading`

[3] Threading and multiprocessing, `http://uwpce-pythoncert.github.io/SystemDevelopment/threading-multiprocessing.html`

[4] Introduction to Parallel and Concurrent Programming in Python, `https://code.tutsplus.com/articles/introduction-to-parallel-and-concurrent-programming-in-python--cms-28612`