# Universidade Federal do ABC
# MCTA016-13 - Paradigmas de Programação
# 2019.Q2

**Aula Prática 10 - K-Means Paralelo**

Prof. Emílio Francesquini

14 de agosto de 2019

A tarefa de agrupamento de dados consiste em particionar uma base de dados de tal forma que cada partição contém elementos que estão próximos entre si no espaço Euclidiano.

O problema das **K-médias** busca por $k$ pontos, denominados centros, com o objetivo de minimizar a soma das distâncias entre cada ponto da base de dados e o centro mais próximo a ele.

A solução para esse problema é feita geralmente de forma heurística utilizando o algoritmo de Lloyd:

- Determine centros iniciais aleatórios
- Associe cada ponto da base com o centro mais próximo, gerando $k$ partições
- Para cada partição, calcule um novo centro como a média dos pontos da partição
- Repita os dois passos anteriores $n$ vezes

Uma versão paralela desse algoritmo pode ser pensada da seguinte forma:

- Determine centros iniciais aleatórios
- Divida os dados em $p$ pedaços, para cada pedaço faça em paralelo:
  - Associe cada ponto do pedaço com o centro mais próximo, gerando $k$ partições
  - Para cada partição, calcule a soma dos pontos associados a ela e a quantidade de pontos
- Junte o resultado de cada pedaço, calculando a média

- Repita os dois passos anteriores $n$ vezes

Para essa tarefa utilizaremos o arquivo `Wine_Quality_DataClean.csv` como base de dados. Esse arquivo possui várias linhas e cada linha contém números do tipo `Double`. Cada linha do arquivo representa um ponto da base de dados. Nossa primeira tarefa é ler esse arquivo e gerar uma matriz do tipo `Double`, ou seja, um `[[Double]]`.

Inicie um novo projeto com `stack new kmeans simple`, no arquivo *cabal* acrescente na seção *executable kmeans*:

```
ghc-options:       -Wall -threaded -rtsopts -with-rtsopts=-N -eventlog -O2
other-modules:     KMeansPar
build-depends:     base >= 4.7 && < 5,
                   parallel,
                   formatting,
                   clock,
                   split,
                   deepseq,
                   array
```

No arquivo `Main.hs` substitua seu conteúdo por:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import System.Environment
import Formatting
import Formatting.Clock
import System.Clock
import Data.List.Split (chunksOf)
import Control.DeepSeq

main :: IO ()
main = do
    [fname, sk, sit, schunks] <- getArgs

    fileTrain <- readFile fname
    let k        = read sk      :: Int
        it       = read sit     :: Int
        nchunks  = read schunks :: Int
        train    = parseFile fileTrain
        chunks   = force $ chunksOf nchunks train
        clusters = take k train
```

```
    start <- getTime Monotonic


    stop <- getTime Monotonic
    fprint (timeSpecs % "\n") start stop
    return ()
```

Crie um arquivo `KMeansPar.hs` com o conteúdo:

```
module KMeansPar where

import Data.List
import Data.Function
import Control.Parallel.Strategies
import Data.Array
import Control.DeepSeq
```

# 1 Leitura do arquivo

A função `readFile` tem a assinatura:

```
readFile :: FilePath -> IO String
```

Ela retorna o conteúdo de um arquivo em forma de `String`. Para transformar esse arquivo em uma lista de lista de `Double`, precisamos fazer:

- Separar cada `\n` em um elemento da lista, gerando um `[String]`
- Dividir cada elemento dessa lista tokenizando a String pelo espaço, gerando um `[[String]]`
- Converter cada elemento dessa lista de lista para o tipo `Double`

As duas primeiras tarefas podem ser feitas utilizando as funções `lines` e `words`. Implemente a função `parseFile` com a seguinte assinatura:

```
parseFile :: String -> [[Double]]
parseFile = ???
```

# 2 Algoritmo K-Means: funções e tipos base

No arquivo **KMeansPar.hs** vamos definir os tipos que trabalharemos, defina `Ponto` e `Cluster` como listas de `Double` e defina um tipo `ChunksOf a` como sendo uma lista de `a`. Ele será utilizado para distribuir as tarefas entre os threads.

Vamos definir também alguns operadores auxiliares:

```haskell
-- divide cada elemento de xs por x
(./) :: (Floating a, Functor f) => f a -> a -> f a
xs ./ x = (/x) <$> xs

-- eleva cada elemento de xs por x
(.^) :: (Floating a, Functor f) => f a -> Int -> f a
xs .^ x = (^x) <$> xs

-- soma dois vetores elemento-a-elemento
(.+.) :: (Num a) => [a] -> [a] -> [a]
(.+.) = zipWith (+)

-- subtrai dois vetores elemento-a-elemento
(.-.) :: (Num a) => [a] -> [a] -> [a]
(.-.) = zipWith (-)

length' :: Num b => [a] -> b
length' xs = fromIntegral $ length xs
```

## 3 Algoritmo K-Means: Associando pontos

Crie uma função `assign` que recebe como parâmetro uma lista de centros, uma lista de pontos e retorna uma array contendo listas de pontos. A Array (da biblioteca `Data.Array`) permite criar uma lista de chave valor cujos valores se acumulam. Exemplo:

```haskell
accumArray (+) 0 (0,4) [(i `mod` 5,i) | i <- [1..10]]
```

Retornará uma lista de tuplas, com cada elemento contendo uma chave de 0 a 4 e o valor para uma chave $k$ será a somatória de todos os $i$'s cujo resultado de $i \mod 5$ retorne $k$. No nosso caso, queremos que a chave seja um inteiro de 0 a $k-1$ e o valor de uma chave $i$ seja a lista de pontos cuja menor distância corresponde ao centro $i$. Ou seja:

```haskell
assign :: [Cluster] -> [Ponto] -> Array Int [Ponto]
assign cs ps = accumArray (flip (:)) [zeroCluster] (0, k-1)
                    [(argmin (distancias p), p) | p <- ps]
  where
    distancias p = ??
    euclid   p c = ??
    argmin    xs = fst $ minimumBy (compare `on` snd) $ zip [0..] xs
    zeroCluster  = replicate (length $ head ps) 0
    k            = length cs
```

Escreva as funções `distancias` e `euclid`. A primeira deve mapear a aplicação da função `euclid p` para cada centro e, a segunda, deve calcular a distância euclidiana entre um ponto `p` e um centro `c`.

## 4 Algoritmo K-Means: Somando os vetores de cada partição

Uma vez que agrupamos os pontos para cada partição, queremos somar esses vetores, resultando em um único vetor e armazenar quantos vetores foram alocados naquela partição. Escreva a função:

```
somaClusters :: Array Int [Ponto] -> [(Cluster, Double)]
```

Para recuperar uma lista de tuplas de uma array, aplique a função `assocs`.

## 5 Algoritmo K-Means: um passo do algoritmo

Cada passo do algoritmo consiste da aplicação de `somaClusters` ao resultado de `assign cs ps`. Complete a função `step`:

```
step :: [Cluster] -> [Ponto] -> [(Cluster, Double)]
step cs ps  =
  -- força avaliação para não deixar o trabalho pra thread principal
  force cs'
  where ???
```

## 6 Algoritmo K-Means: função principal

Finalmente a função principal `kmeans` recebe como parâmetros a quantidade de iterações `it`, pedaços de base de dados `pss`, uma lista de clusters iniciais `cs` e retorna uma lista de `Clusters`. Utilizando Pattern Matching controlamos as repetições dos passos do algoritmo:

```
kmeans :: Int -> ChunksOf [Ponto] -> [Cluster]
       -> [Cluster]
kmeans it pss cs
  | it <= 0   = cs
  | otherwise = kmeans (it-1) pss cs'
  where
    cs'              = ???
```

O novo cluster `cs'` deve ser construído pelos passos:

- Mapear a função `step` em cada pedaço de pontos em paralelo utilizando `rseq`, resultando em uma lista de lista de tuplas
- Somar o resultado dessa aplicação, somando os elementos das tuplas
- Filtrar o resultado eliminando as listas vazias
- Calcular a média em cada elemento da lista, resultando em uma lista de `Cluster`

# 7 Finalizando

No arquivo `Main.hs` acrescente a função principal:

```
start <- getTime Monotonic

let clusters' = kmeans it chunks clusters
print clusters'

stop <- getTime Monotonic
```

Compile o programa com `stack build --profile` e execute com `stack exec kmeans -- Wine_Quality_DataClean.csv 3 1000 1000 +RTS -N1`

Execute e compare o algoritmo utilizando diferentes valores para o argumento `-N`.