Clustering

Gustavo Alves Pacheco*

11821ECP011

1 Introdução

Até o momento, as redes neurais eram treinadas de forma supervisionada, ou seja, os dados de treinamento possuiam rótulos, os quais apresentavam, à máquina, a solução esperada, dado um conjunto de entradas. Entretanto, nem sempre os dados de entrada da rede neural estão rotulados. Para isso, o aprendizado não supervisionado entra em questão. Dentre suas aplicações estão:

- Data Mining
- Reconhecimento de padrões
- Compressão de dados

Esta técnica consiste na divisão de um conjunto D, composto por N vetores, em k grupos, de tal forma que os centros desses agrupamentos sejam localizados visando a minimização da variância inter-classe [1].

O algoritmo que será tratado neste relatório é o do *k-means*. Neste algoritmo, um número de centroides igual ao número de agrupamentos desejado é inicializado aleatoriamente. Esses centroides dividem o grupo por uma reta que se encontra entre os pares de centros. A partir daí, a distância euclidiana entre cada ponto e o centroide correspondente é calculada, e a posição do segundo é atualizada, de acordo com a média dessas distâncias de cada grupo, até que não haja mais migração de pontos para outro cluster.

Entretanto, este algoritmo apresenta alguns problemas, envolvendo principalmente tempo de execução muito grande, além do fato que a solução encontrada pode ser ruim e distante do agrupamento ótimo. Para resolver isso, *k-means++* inicializa aleatoriamente o primeiro centroide. A partir dele, calcula a distância entre ele e x_i. Assim, a próxima centroide é escolhida aleatoriamente usando uma distribuição ponderada de probabilidade onde um ponto x_i é escolhido com probabilidade proporcional ao quadrado da distância $D(x_i)$. Esse processo se repete até que todas as centroides iniciais sejam definidas.

*gap1512@gmail.com

2 Objetivos

- Aprimorar o conhecimento sobre Redes Neurais Artificiais e obter experiência prática na implementação das mesmas.
- Encontrar o agrupamento adequado para os dados apresentados no arquivo inputs.points usando o algoritmo *k-means* clássico.
- Implementar *k-means++* para realizar o agrupamento no mesmo arquivo.
- Levantar a curva do erro quadrático total para cada caso.

3 Materiais e Métodos

Para implementação do algoritmo de clustering foi utilizada a linguagem de programação Common Lisp, compilando-a com o SBCL (Steel Bank Common Lisp). Como interface de desenvolvimento, foi utilizado o Emacs em Org Mode, configurado com a plataforma SLIME (The Superior Lisp Interaction Mode for Emacs) para melhor comunicação com o SBCL. Foi utilizada uma abordagem bottom-up para o desenvolvimento. O código produzido segue majoritariamente o paradigma funcional, sendo este trabalho como um todo uma obra de programação literária. Parte das funções já foram implementadas em Regra de Hebb, Perceptron e Adaline, Regressão Linear, Multilayer Perceptron e Feature Engineering.

4 Desenvolvimento

Iniciando pela leitura dos pontos, a mesma pode ser feita da seguinte forma:

```
(defvar *inputs-clustering*
  (read-csv #p"data/inputs.points" :separator '(#\ )))
```

Assim, os pontos da lista são representados pela figura 1.

```
(scatter-plot "plots/inputs.png" *inputs-clustering*
              nil 0 0 "black" "black")
```

A partir daí, começa-se a implementação do algoritmo *k-means* clássico e em seguida, do *k-means++*. Seguindo a metodologia utilizada até o momento, as funções primitivas serão construídas, e depois agrupadas para geração das funções mais complexas.

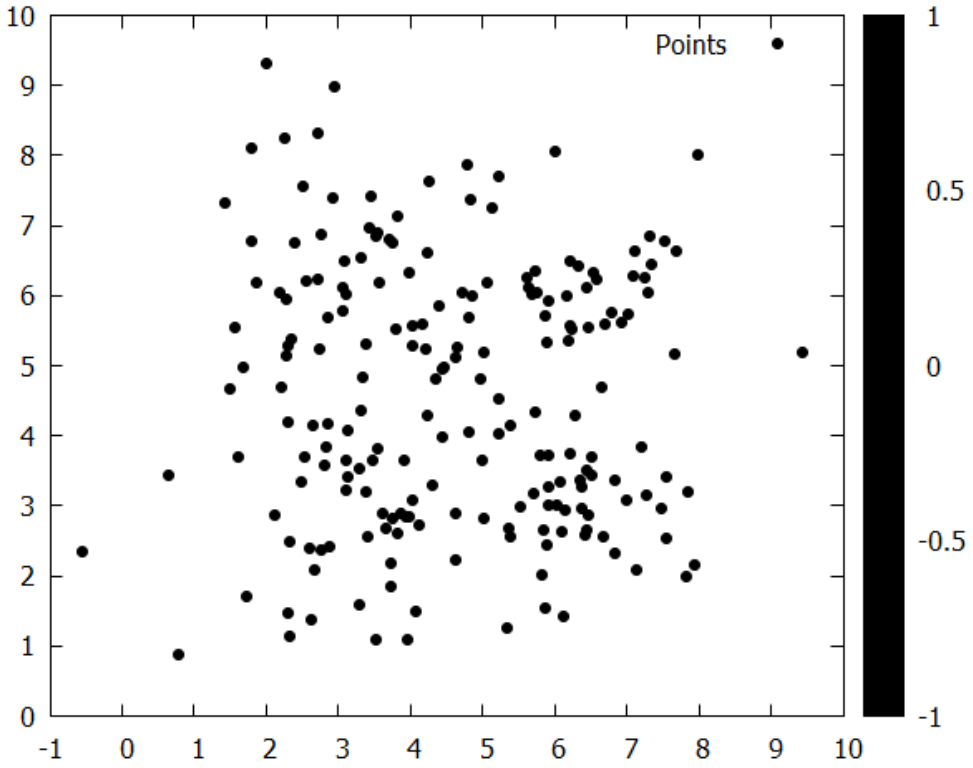

Figure 1: Pontos do arquivo de entradas

```
(defun initial-centers (k points)
  (let ((lt (length points)))
    (loop repeat k collecting
         (nth (random lt) points))))
```

Para o cálculo da distância euclidiana:

```
(defun euclidean-distance (point-1 point-2)
  (sqrt (reduce #'+ (mapcar #'(lambda (p q)
                                (expt (- p q) 2))
                            point-1 point-2))))
```

Assim, dada uma lista de centroides e um ponto, a determinação do grupo ao qual faz parte é a seguinte:

```
(defun group-point (point centers)
  (labels ((rec (lst i result)
             (if lst
                 (rec (cdr lst)
                      (1+ i)
                      (let ((dist (euclidean-distance
                                   point
                                   (car lst))))
                        (if (< dist (first result))
                            (list dist i)
                            result)))
                 result)))
    (rec (cdr centers) 1
         (list (euclidean-distance point (car centers)) 0))))
```

Para atualização de um centro, utiliza-se a definição da média, da seguinte maneira:

```
(defun update-centers (k points-pos)
  (loop for i from 0 upto (1- k)
     collect (let ((lst (remove-if-not
                         #'(lambda (p)
                             (eq (first (last p)) i))
                         points-pos)))
               (if lst
```

```
                    (butlast (multiple-value-list
                              (average lst)))
                    '(0 0)))))
```

E assim, a função *k-means* é implementada:

```
(defun k-means (k points initial-centers max-iterations tolerance)
  (do* ((distances nil (mapcar #'(lambda (p)
                                   (group-point p centers))
                               points))
        (old-err nil err)
        (err nil (reduce #'+ distances :key #'first))
        (b nil (mapcar #'second distances))
        (errs nil (cons (list i err) errs))
        (points-pos nil (mapcar #'(lambda (p b)
                                    (append p (list b)))
                                points b))
        (centers initial-centers (update-centers k points-pos))
        (i 0 (1+ i)))
       ((or (> i max-iterations)
            (and (> i 1)
                 (<= (- old-err err) tolerance)))
        (values centers points-pos errs err))))
```

A chamada da mesma é algo do tipo (figs. 2 3):

```
(let ((k 4))
  (multiple-value-bind (centers points errors)
      (k-means k *inputs-clustering*
               (initial-centers k *inputs-clustering*) 300 0)
    (scatter-plot "plots/clusters-k-means.png" points
                  nil 0 3 "yellow" "magenta")
    (scatter-plot "plots/quadratic-error-k-means.png" errors
                  nil 0 0 "black" "black")
    centers))
```

Os centros encontrados para 4 grupos são:

3.100963	2.8521535
2.9818652	6.347798
6.2758636	3.0102816
6.11724	6.039169

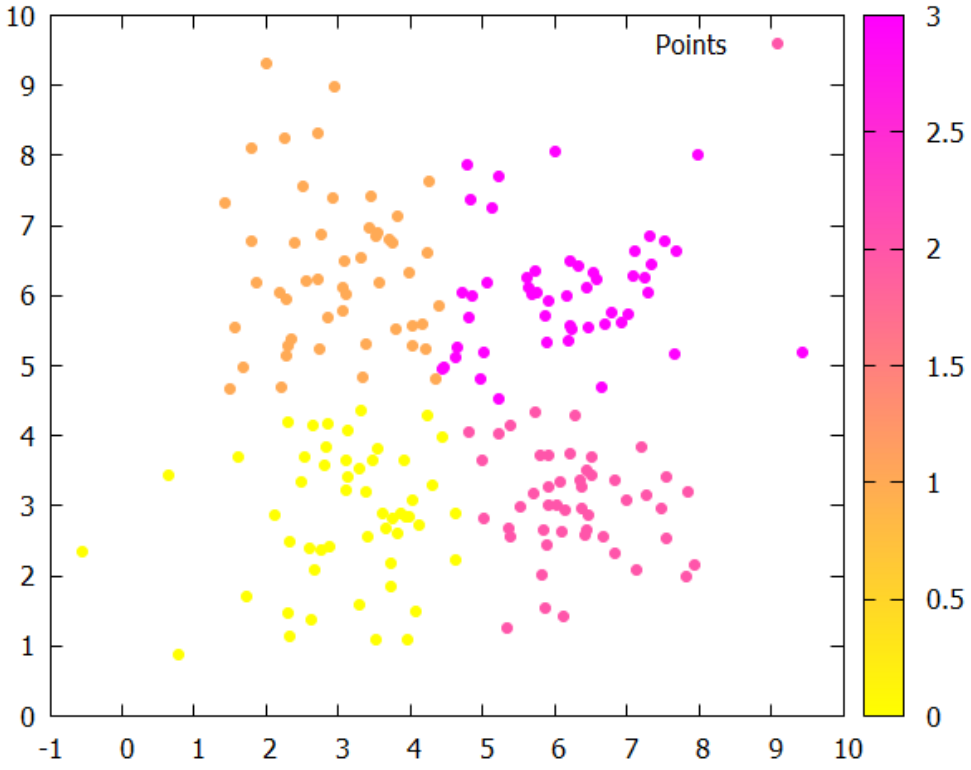

Figure 2: Agrupamento em 4 Clusters utilizando *k-means* clássico

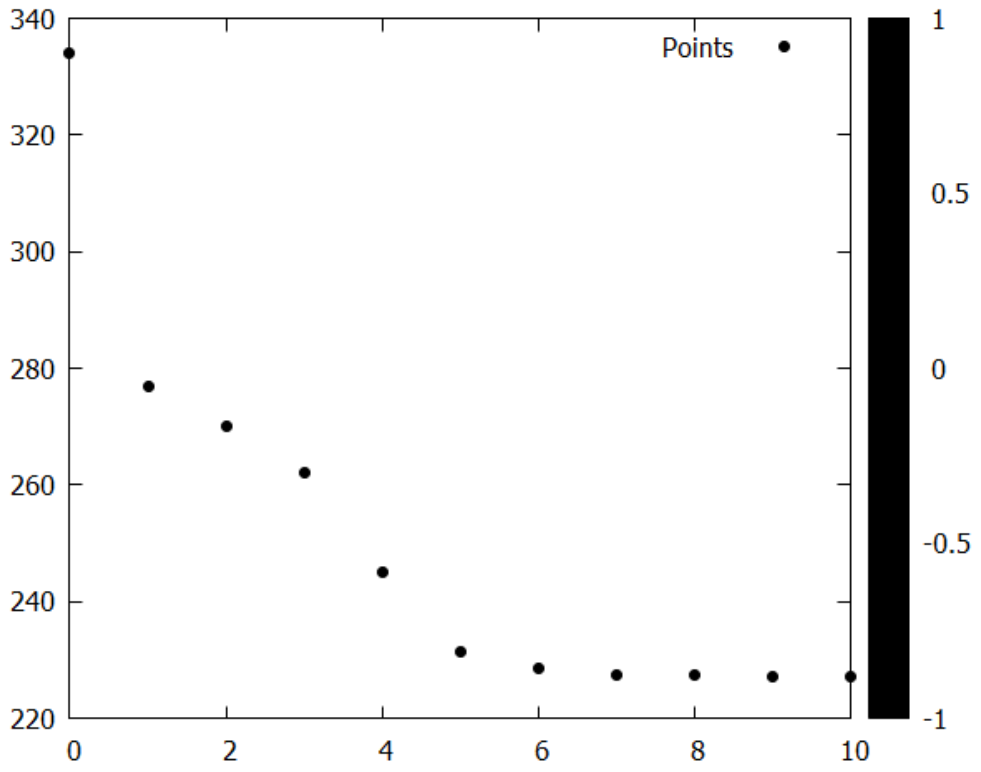

Figure 3: Erro quadrático por iteração do agrupamento em 4 Clusters (*k-means* clássico)

A diferença do *k-means* clássico para o *k-means++* está na geração inicial dos valores. Assim, não é necessário modificar as funções acima, apenas implementar uma nova geradora de centroides iniciais. Para isso, a função da roleta é implementada:

```
(defun search-proc (c-point rnd)
  (< (first c-point) rnd (second c-point)))

(defun d-point (point distance)
  (cons distance point))

(defun c-points (d-points)
  (do ((sum (reduce #'+ d-points :key #'first))
       (points d-points (cdr points))
       (aux 0 acc)
       (acc 0)
       (res nil (append res
                        (list (cons aux
                                    (cons acc
                                          (cdr (car points))))))))
      ((not points) res)
    (setf acc (+ acc (/ (first (car points)) sum)))))

(defun roulette (c-points)
  (let ((rnd (random 1.0)))
    (nthcdr 2 (find-if #'(lambda (x)
                           (search-proc x rnd))
                       c-points))))
```

O teste seria algo do tipo:

```
(roulette (c-points (mapcar #'d-point '((1 2) (3 4) (5 6)) '(1 2 3))))
```

5 6

Nessa chamada, uma lista é criada através do mapeamento da função `d-point` nas lista de pontos e distâncias correspondentes. Essa lista é apenas a inclusão da distância como primeiro elemento dos pontos. Assim, quando a função `c-points` é invocada, a mesma transforma essa distância em um valor cumulativo proporcional, que vai de zero a um. Esse valor indica a faixa que cada ponto ocupa na roleta. Assim, `roulette` apenas utiliza de `search-proc` para encontrar o ponto escolhido aleatoriamente. Desta forma, *k-means++* é implementada:

```
(defun k-means++ (k points)
  (do ((i 1 (1+ i))
       (centers (initial-centers 1 points)
                (cons (roulette
                       (c-points
                        (mapcar #'d-point
                                points
                                (lesser-distance points centers))))
                      centers)))
      ((>= i k) centers)))

(defun lesser-distance (points centers)
  (mapcar #'first (mapcar #'(lambda (p)
                              (group-point p centers))
                          points)))
```

A chamada da mesma é:

```
(k-means++ 4 '((1 2) (2 3) (3 4) (4 5)))
```

3	4
2	3
4	5
1	2

Assim, o clustering é feito (figs. 4 5):

```
(let ((k 4))
  (multiple-value-bind (centers points errors)
      (k-means k *inputs-clustering*
               (k-means++ k *inputs-clustering*) 300 0)
    (scatter-plot "plots/clusters-k-means++.png" points
                  nil 0 3 "yellow" "magenta")
    (scatter-plot "plots/quadratic-error-k-means++.png" errors
                  nil 0 0 "black" "black")
    centers))
```

Os centros encontrados para 4 grupos são:

6.2758636	3.0102816
2.9818652	6.347798
3.100963	2.8521535
6.11724	6.039169

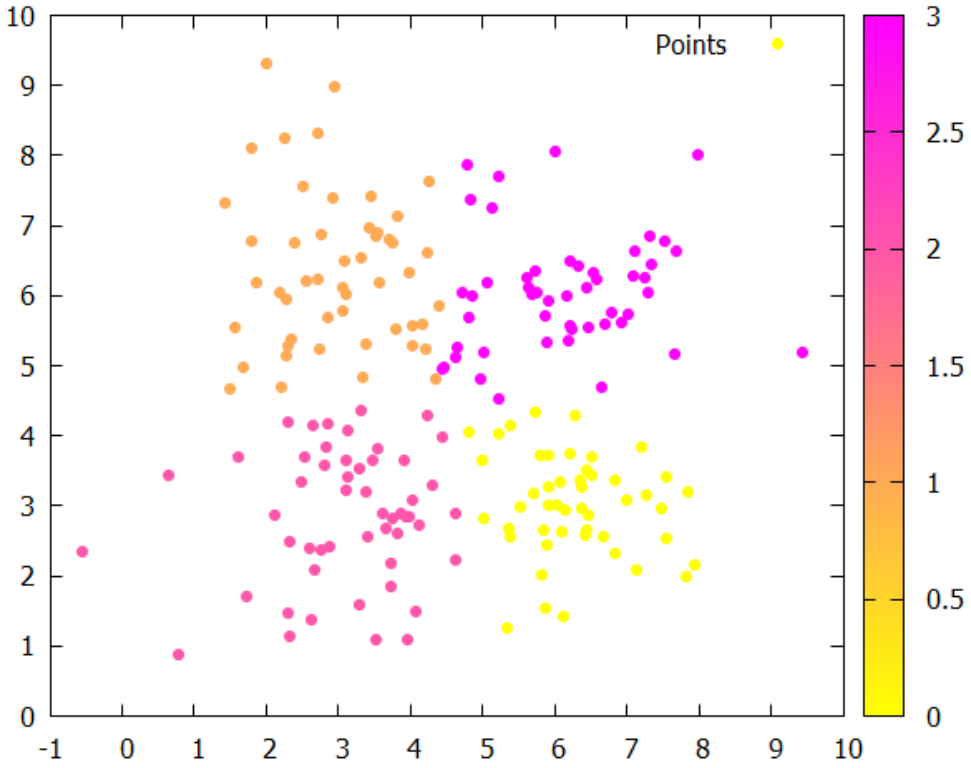

Figure 4: Agrupamento em 4 Clusters utilizando *k-means++*

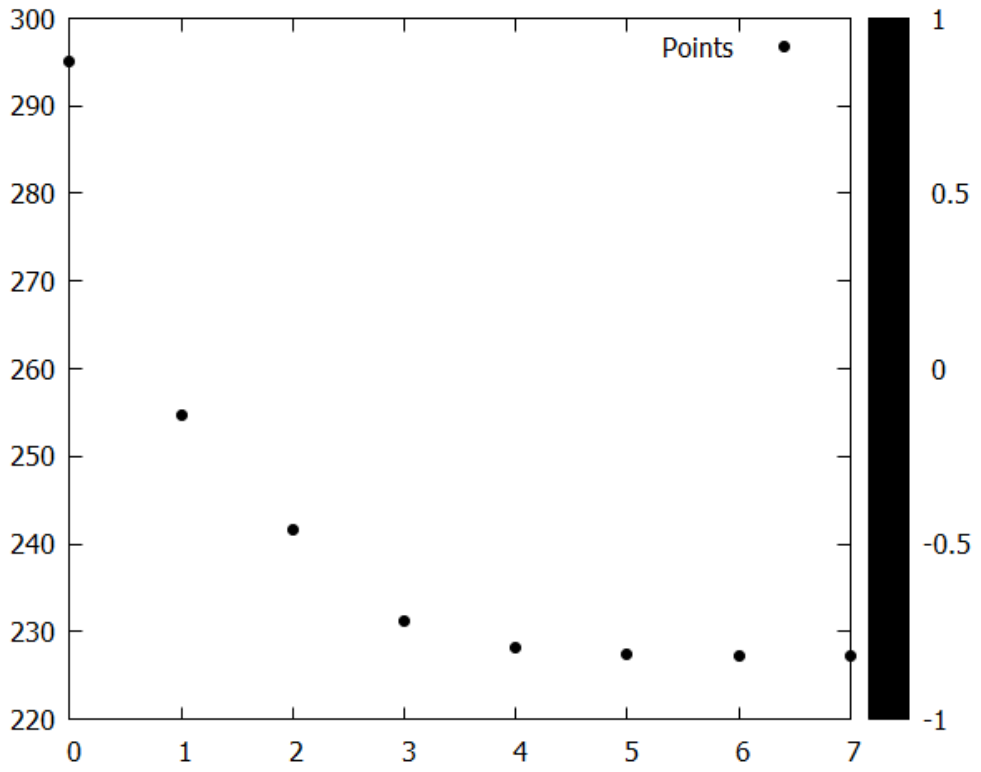

Figure 5: Erro quadrático por iteração do agrupamento em 4 Clusters (*k-means++*)

Por fim, é definida a função que faz o clustering sem que se saiba o valor de k. Para isso, o mesmo é incrementado, e o erro total é comparado a cada iteração. Quando a diferença entre gerações for menor que um valor especificado, a execução termina.

```
(defun clustering (points generation-fn max-iterations-per-cycle
                   tolerance-per-cycle max-iterations tolerance)
  (do ((i 1 (1+ i))
       (errs-k nil (cons (list i err) errs-k))
       centers points-pos errs err)
      ((or (>= i max-iterations)
           (and (> i 2)
                (< (- (cadadr errs-k) (cadar errs-k))
                   tolerance)))
       (values centers points-pos errs err (nreverse errs-k) i))
    (setf (values centers points-pos errs err)
          (k-means i points (funcall generation-fn i points)
                   max-iterations-per-cycle tolerance-per-cycle))))
```

Na execução a seguir, o valor máximo de grupos é 100, e a execução termina quando a diferença do erro entre os grupos for nula. As figuras 6 e 7 abaixo apresentam o agrupamento em N grupos e a fig. 8, o erro a cada incremento de k.

```
(multiple-value-bind (centers points errors err errs-k max-k)
    (clustering *inputs-clustering* #'k-means++ 5000 0 100 0)
  (scatter-plot "plots/clusters-multiple-k.png" points
                nil 0 3 "yellow" "magenta")
  (scatter-plot "plots/quadratic-error-multiple-k.png" errors
                nil 0 0 "black" "black")
  (scatter-plot "plots/error-by-k.png" errs-k
                nil 0 0 "black" "black")
  (values centers err max-k))
```

E estes foram os centros dos grupos encontrados com o valor de k (ver fig. 8 para conhecer o número de grupos) que encerrou a execução:

3.3839722	6.5847006
3.4072819	4.742861
6.0601697	2.5497687
6.6205506	3.8007708
6.301741	6.209492
2.177675	8.075488
2.957582	2.537351

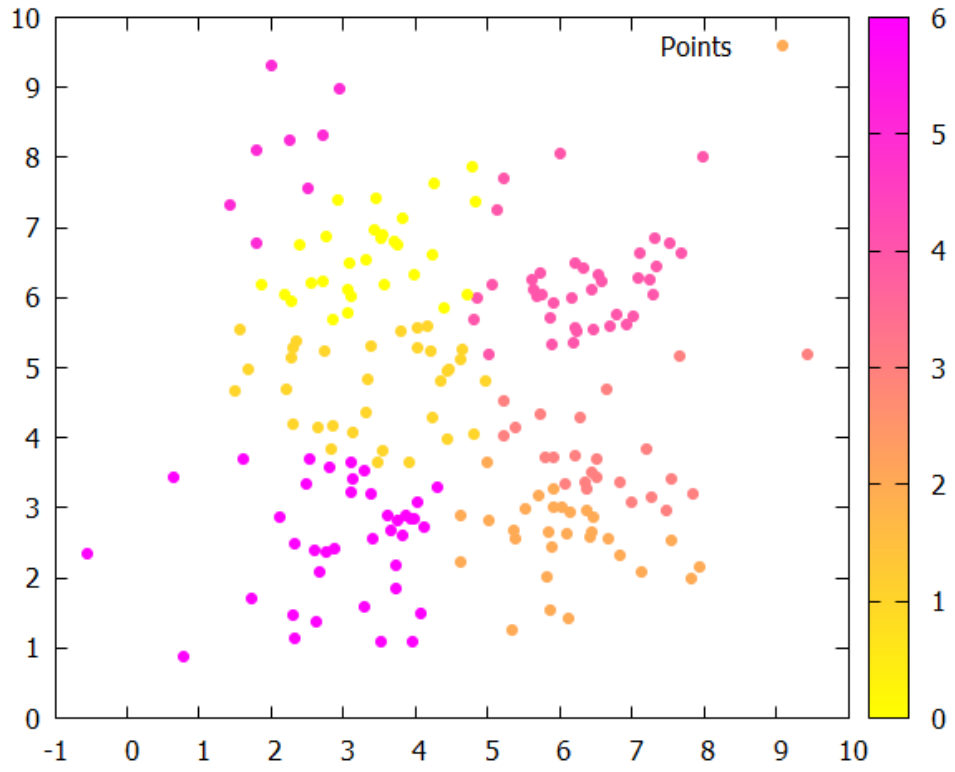

Figure 6: Agrupamento em N Clusters utilizando *k-meanqs++* incremental

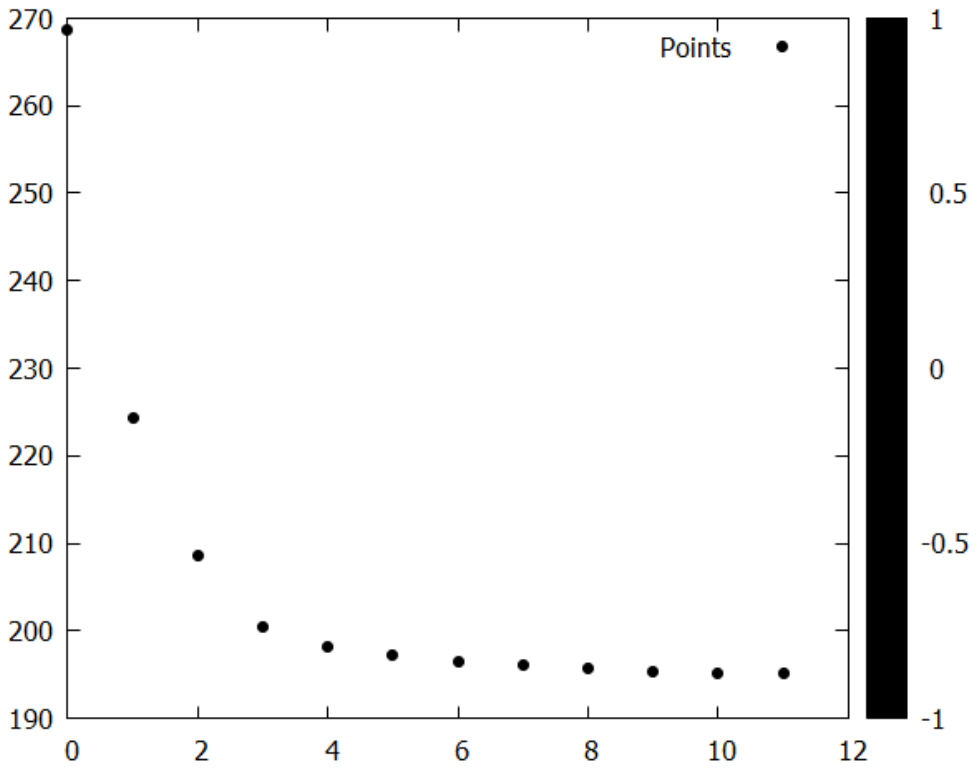

Figure 7: Erro quadrático por iteração do agrupamento em N Clusters (*k-means++* incremental)

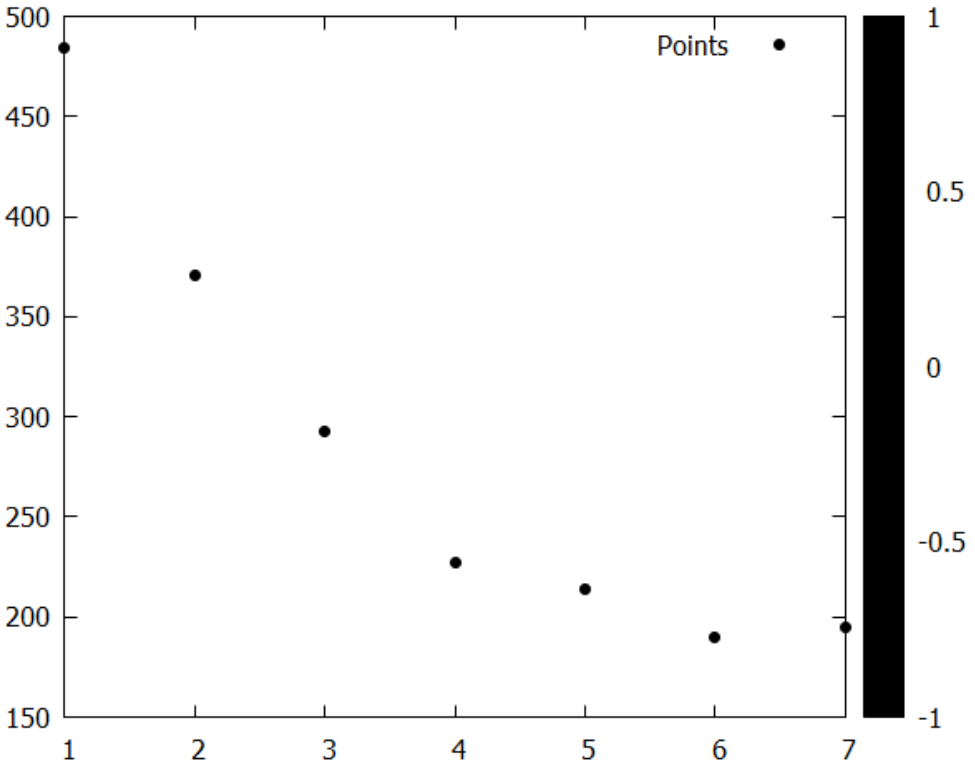

Figure 8: Erro quadrático por incremento de k (*k-means++* incremental)

5 Conclusão

Desta maneira, foi possível dividir os pontos em uma quantidade arbitrária de grupos, além da ter sido possível a comparação entre os dois algoritmos de inicialização. Após várias execuções, notou-se que o *k-means++* encontrava a solução mais rapidamente, se comparado ao clássico. Vale observar que nem sempre a afirmação é verdadeira, visto que em alguns casos a situação se inverteu. No algoritmo que incrementa o número de grupos, observou-se que uma diferença de erro entre diferentes k's nula ocorria quando a divisão era feita com aproximadamente 8~15 grupos.

Um experimento interessante a se fazer é o incremento do k até o tamanho do conjunto de entradas. Dessa forma (fig. 9):

```
(multiple-value-bind (centers points errors err errs-k max-k)
    (clustering *inputs-clustering* #'k-means++ 5000 0 (length
    ↪  *inputs-clustering*) -10)
  (scatter-plot "plots/error-by-k-length.png" errs-k
                nil 0 0 "black" "black")
  (values centers points errors err max-k))
```

É possível ver que a inclinação muda em aproximadamente 20 grupos (fig. 9), que deveria ser a parada do algoritmo. E como esperado, com k igual ao tamanho do grupo, o erro é zero, pois cada entrada passa a ser um centroide. Em valores de k iguais a 8, 15 e 21, geralmente encontra-se pontos que apresentam o mesmo erro, por isso a parada citada anteriormente ocorria nessa faixa.

References

[1] K. Yamanaka. Aprendizagem de máquina (machine learning - ml). Universidade Federal de Uberlândia.

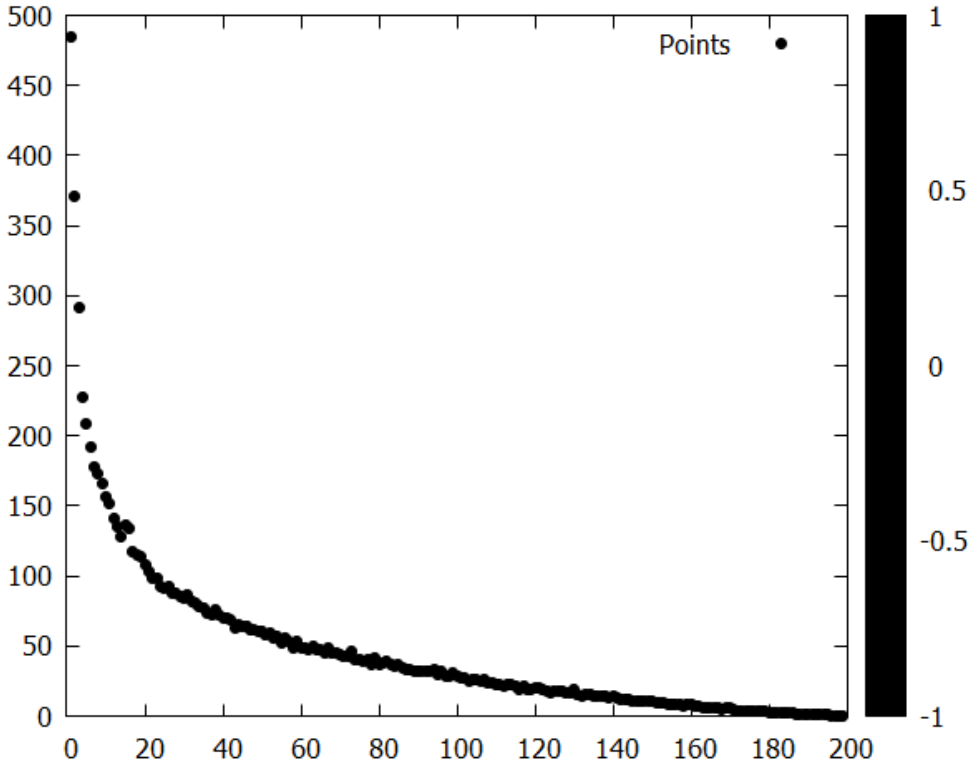

Figure 9: Incremento de grupos até o tamanho da entrada