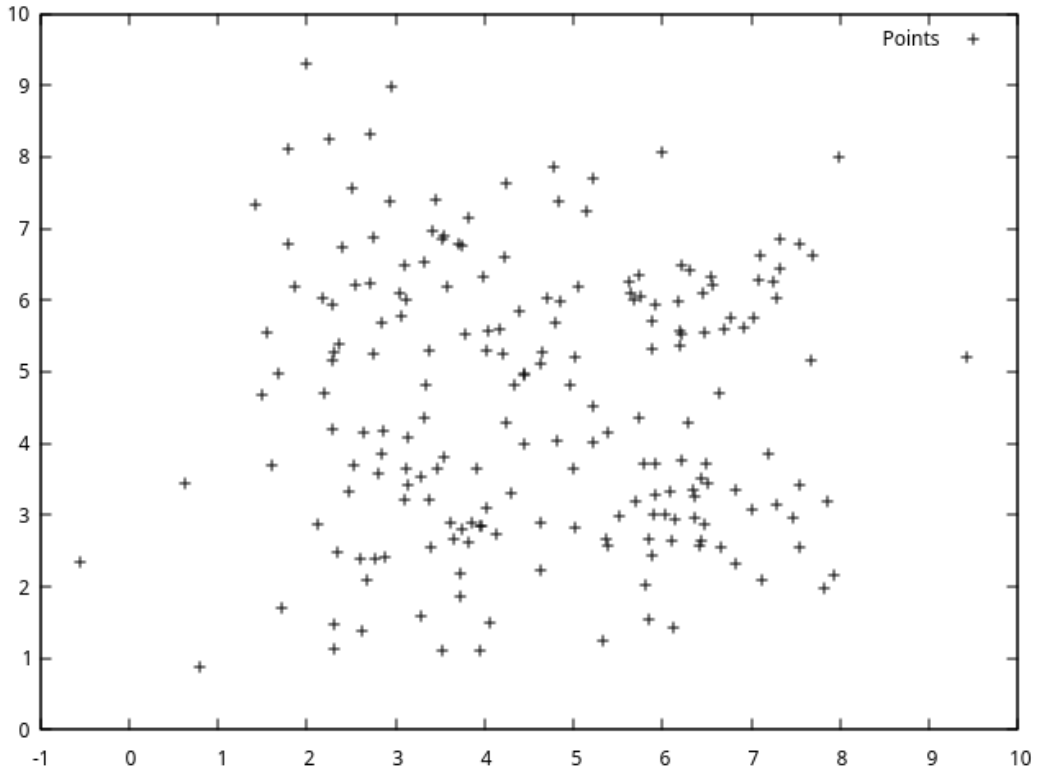
Relatório 7 - Agrupamento

1 Introdução

Esse relatório foi feito com base no arquivo *observacoes.txt*, no relatório chamado de *dados*, provido pelo professor. Nele apresentam-se os seguintes pontos:



O aprendizado não supervisionado consiste em um treinamento no qual os dados de treinamento não são rotulados. Ele é utilizado para encontrar e agrupar dados similares e, posteriormente, rotular os dados pertencentes aos agrupamentos encontrados. Ele possui aplicações em data mining, reconhecimento de padrões e compressão de dados.

Durante o processo de agrupamento dos dados deseja-se agrupar os vetores em grupos de maneira que o agrupamento seja ótimo, para isso é necessário localizar centros de agrupamentos que minimizem a variância inter-classe. Os problemas que aparecem são como encontrar esses centos? E qual a quantidade ideal de centros?

O algoritmo k-means foi criado com o intuito de ajudar a resolver esse problemas. Ele inicia um número de centroides com valores aleatórios e usa da distância euclidiana para calcular a distância entre os pontos e os centroides. A cada iteração ele atualiza a posição de cada centroide calculando a média de cada atributo dos pontos que pertencem a cada agrupamento. Ele continua iterando até que não haja mais alteração na posição dos centroides.

Apenas do k-means ser um algoritmo muito poderoso ele tem alguns problemas, como que seu tempo de execução pode ser muito grande e a solução encontrada pode ser ruim e distante do agrupamento ótimo. Foi com intuito de resolver esses problemas que o k-means++ foi criado. Ele adiciona alguns passos adicionais ao k-means, agora inicia-se os valores dos centroides com valores aleatórios dentro dos pontos de entrada, depois calcula-se a distância entre os centroides e os valores de entrada e escolhe-se outro valor com probabilidade proporcional ao quadrado da distância do valor ao centroide. Finalmente, executa-se o k-means usando os valores de centroides encontrado no k-means++ como valores iniciais das centroides.

Nesse relatório foram utilizados ambos o k-means e o k-means++ para agrupar os dados do arquivo fornecido pelo professor:

2 Objetivo

1. Encontrar o agrupamento adequado para os *dados* apresentados no arquivo dados usando o algoritmo k-means clássico.
2. Implementar k-means++ para realizar o agrupamento no mesmo arquivo do item 1.
3. Levantar a curva do erro quadrático total para cada caso.

3 Desenvolvimento

Para realizar a implementação dos dois algoritmos foi utilizada a linguagem de programação Common Lisp com um paradigma primordialmente funcional. Como dessa vez foi implementado um algoritmo de aprendizado não supervisionado a única função aproveitada dos trabalhos anteriores foi a de leitura do arquivo com algumas alterações. Como a função anterior separava os valores da linha usando a virgula e nessa semana o arquivo de dados apresenta duzentas linhas de valores de x e y separados por dois espaços foi necessário realizar mudar a função para que essa separasse as linhas por espaço. Todavia, como dito anteriormente, os valores no arquivo estão separados por dois espaços, foi necessário, portanto, tratar o arquivo excluindo um desses espaços a fim de que as seguintes funções pudessem lê-lo:

```
(defun file-to-list (filename)
  (with-open-file (stream filename)
```

```
    (loop for line = (read-line stream nil)
       while line
       collect (mapcar #'string-first (uiop::split-string line
       ↪  :separator '(#\ ))))))
(defun string-first (string)
  (with-input-from-string (str string) (read str nil nil)))
```

Com a leitura do arquivo pronta foi hora de começar a implementar o k-means, entretanto antes de começar foi necessário criar algumas funções que seriam utilizadas por ele. Primeiramente foi implementada a função de criação dos k's. Essa função pega um valor aleatório pra x e y entre 0 e 10:

```
(defun make-random-k (number)
  (labels ((mk-rdm (num list)
             (cond
               ((eql 0 num) list)
               (t (mk-rdm (1- num) (append list (list (list (random 10)
               ↪  (random 10)))))))))
    (mk-rdm number '())))
```

Com essas funções foram criadas as seguintes variáveis para facilitar o processo de teste das funções:

```
(defvar data (file-to-list "./class/data"))
(defvar list-of-k (make-random-k data 5))
```

Foi criada então uma função para calcular a distância entre dois pontos a fim de criar a função que cria a lista b que dita a centroide mais próxima de cada entrada:

```
(defun distance-points (p1 p2)
  (sqrt (+ (expt (- (first p1) (first p2)) 2) (expt (- (second p1)
  ↪  (second p2)) 2))))

;; Creates the b list
;; List of lists, list of lists -> list
(defun make-b (data list-of-k)
  (labels ((find-fitting-k (d b)
             (cond
               ((null d) b)
               (t (find-fitting-k
```

```
                    (rest d)
                    (append b (list
                               (loop
                                  for i from 0 upto (1- (list-length
                                  ↪  list-of-k))
                                  when (eql
                                        (distance-points (nth i
                                        ↪  list-of-k) (first d))
                                        (apply #'min (mapcar #'(lambda
                                        ↪  (k) (distance-points (first
                                        ↪  d) k)) list-of-k)))
                                  do (return (1+ i))))))))))
    (find-fitting-k data '())))
```

Com a lista b pronta foi hora de criar a função de ajuste dos centroides, para facilitar esse ajuste foi criada também a função de agrupamento da data que agrupava os dados em listas correspondentes aos seus centroides:

```
;; Agrupates the data using b
;; List of lists, list of lists, list -> list of lists
(defun agrupate-data (data list-of-k list-of-b)
  (labels ((foo (num result)
             (cond
               ((eql num (1+ (list-length list-of-k))) result)
               (t (foo
                   (1+ num)
                   (append result (list
                                   (labels ((foo2 (d b list)
                                              (cond
                                                ((null d) list)
                                                (t (foo2 (rest d) (rest
                                                ↪  b) (if (eql (first
                                                ↪  b) num) (append
                                                ↪  list (list (first
                                                ↪  d))) list))))))
                                     (foo2 data list-of-b '()))))))))))
    (foo 1 '())))

;; Ajusts the k's to the new values
;; List of lists, list of lists, list -> list of lists
(defun ajust-k (data list-of-k list-of-b)
  (let* ((agrupated-data (agrupate-data data list-of-k list-of-b)))
```

```
    (mapcar #'(lambda (d k)
                (if (not (null d)) (list
                                    (/ (apply #'+ (mapcar #'first d))
                                    ↪  (list-length d))
                                    (/ (apply #'+ (mapcar #'second d))
                                    ↪  (list-length d)))
                    k))
            agrupated-data list-of-k)))
```

Com o ajuste dos centroides pronto bastou fazer a criação da função k-means que retorna os valores finais dos centroides, a lista de b's e uma lista dos erros, para calcular esse erros foi necessário criar a função de squared-error, também apresentada abaixo:

```
;; Calculates the squared error
;; List of lists, list of lists -> number
(defun squared-error (agrupated-data list-of-k)
  (let* ((list-of-errors
          (mapcar #'(lambda (x) (expt x 2))
                  (apply #'append (distance-k agrupated-data
                  ↪  list-of-k)))))
   (apply #'+ list-of-errors)))

;; Calculates the final values of K
;; List of lists, list of lists -> list of lists, list
(defun k-means (data list-of-k)
  (labels ((condition-test (k error)
             (let* ((list-of-b (make-b data k)) (agrupated-data
             ↪  (agrupate-data data k list-of-b)))
               (if (not (equal k (ajust-k data k list-of-b)))
                   (condition-test (ajust-k data k list-of-b) (append
                   ↪  error (list (squared-error agrupated-data k))))
                   (values k list-of-b error)))))
    (condition-test list-of-k '())))
```

Foi hora, então de implementar o k-means++, para isso foi criada a função que escolhe aleatoriamente um dos dados como ponto inicial do centroide:

```
;; Choses a number of entrances
;; List of lists, number -> List of lists
(defun make-random-k++ (data num)
  (loop
```

```
    for i upto (1- num)
    collect (nth (random (1- (list-length data))) data)))
```

Após isso foi criada a função que escolhe o centroide baseado em uma lista de distancias dele para os outros dados, para que isso fosse possível foi necessário criar uma função que gerasse essa lista ponderada pela distancia:

```
;; List, list -> list
(defun ponderate (data distances)
  (apply #'append (mapcar #'(lambda (d dist) (make-list (round (expt (*
  ↪  10 dist) 2)) :initial-element d))
         data distances)))

;; Choses the k's for the k-means++
;; List of lists, list of lists, list -> list of lists
(defun chose-k++ (data list-of-k)
  (let*
      ((list-of-b (make-b data list-of-k))
       (agrupated-data (agrupate-data data list-of-k list-of-b))
       (distance (distance-k agrupated-data list-of-k)))
    (mapcar #'(lambda (d dist)
                (nth (random (1- (list-length (ponderate d dist))))
                     (ponderate d dist)))
            agrupated-data distance)))
```

Por fim, foi hora de criar as funções para plotar, uma para plotar os pontos ajustados aos seus centroides e outra para plotar o gráfico do erro quadrático:

```
(defun plot (data list-of-k)
  (multiple-value-bind (k b) (k-means data list-of-k)
    (let* ((agrupated-data (agrupate-data data k b)))
      (vgplot:plot
       (mapcar #'first (first agrupated-data)) (mapcar #'second (first
       ↪  agrupated-data)) "r+; k1"
       (mapcar #'first (second agrupated-data)) (mapcar #'second
       ↪  (second agrupated-data)) "b+; k2"
       (mapcar #'first (third agrupated-data)) (mapcar #'second (third
       ↪  agrupated-data)) "g+; k3"
       (mapcar #'first (fourth agrupated-data)) (mapcar #'second
       ↪  (fourth agrupated-data)) "c+; k4"
       (mapcar #'first (fifth agrupated-data)) (mapcar #'second (fifth
       ↪  agrupated-data)) "b+; k5"
```

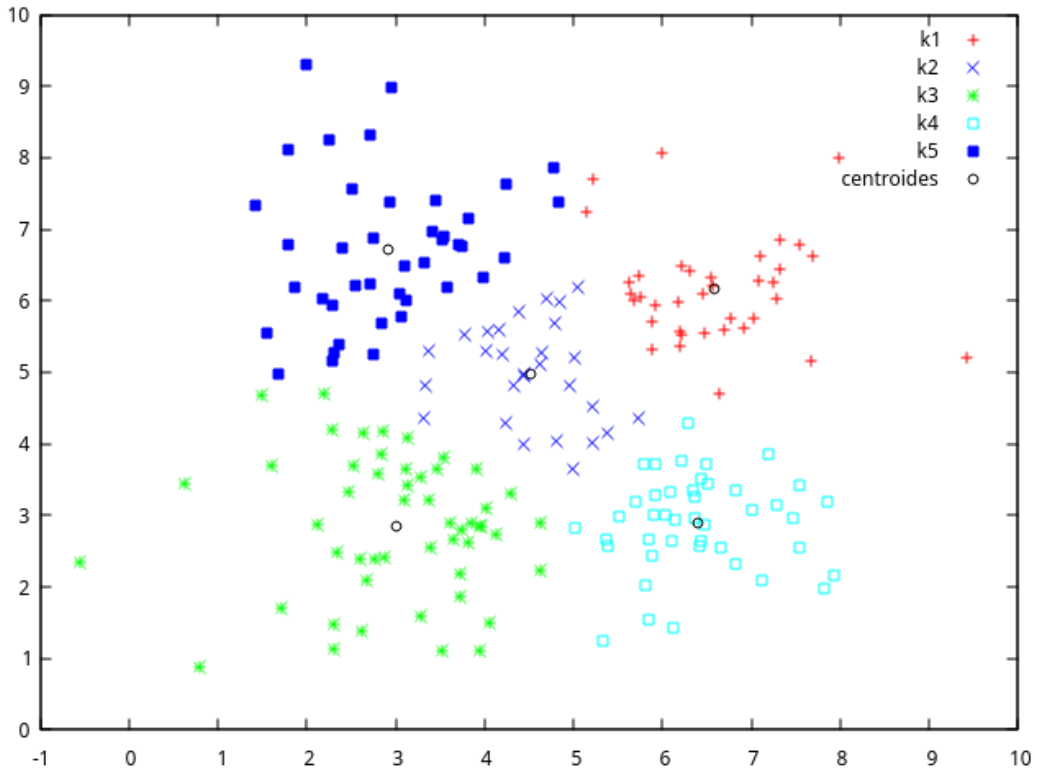
```
      (mapcar #'first k) (mapcar #'second k) "k+; centroides"))))
(defun plot-error (err)
  (vgplot:plot (loop for i from 0 upto (1- (list-length err)) collect
  ↪  i) err "r+; Squared error"))
```

Que produziram os gráficos encontrados na sessão de resultados.

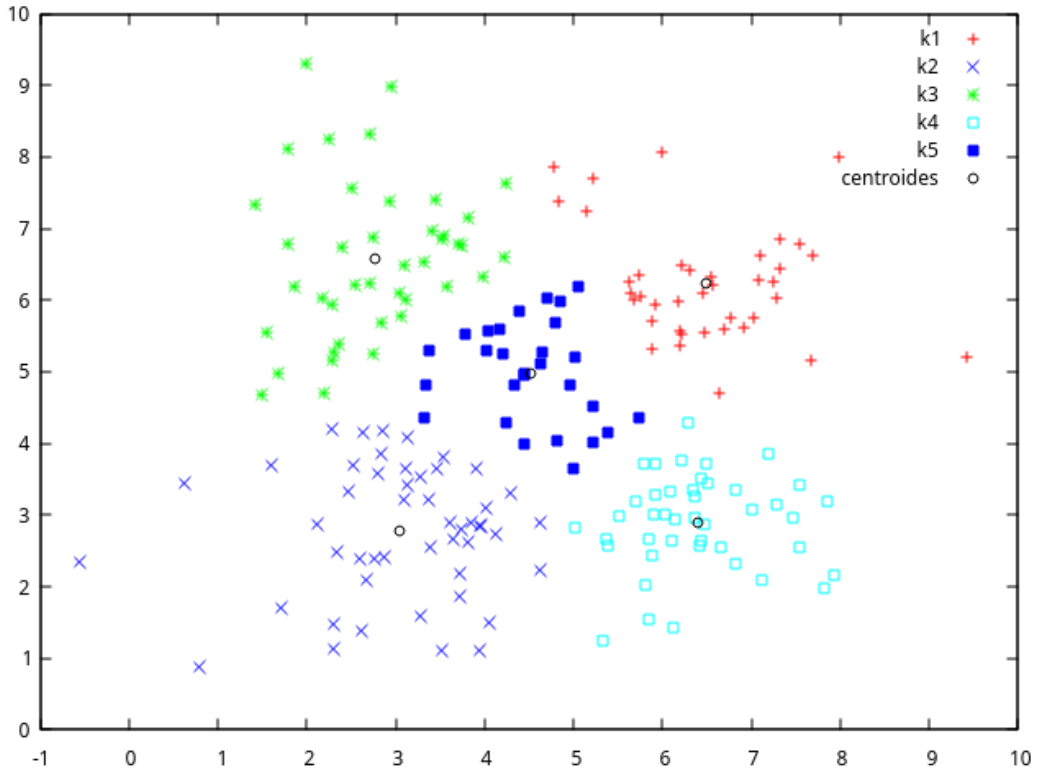
4 Resultados

Os resultados apresentados foram calculados com um número de centroides igual a 5 e com valores iniciais de centroides aleatórios de 0 a 10 para k-means e iguais a uma entrada aleatória para k-means++.

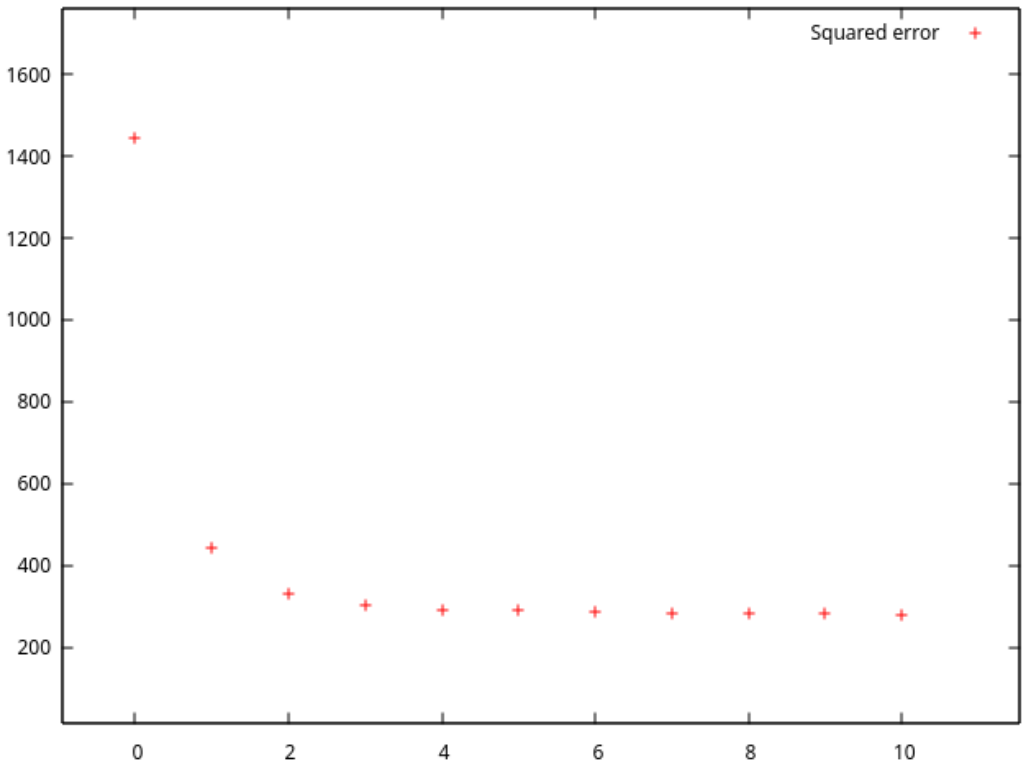
4.1 Gráfico dos pontos para k-means



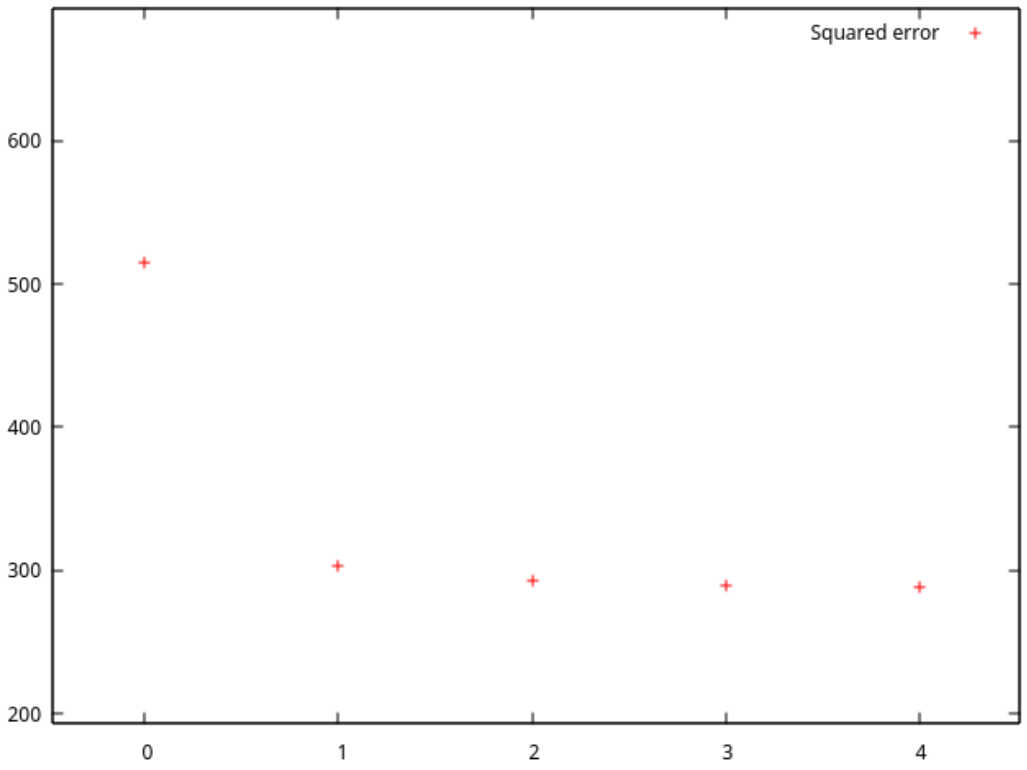
4.2 Gráfico dos pontos para k-means++



4.3 Gráfico dos erros quadráticos para k-means



4.4 Gráfico dos erros quadráticos para k-means++



5 Conclusão

Foi possível, portanto, observar o funcionamento dos algoritmos de k-means e k-means++, que são muito poderosos e ainda utilizados por empresas para fazer o agrupamento de dados. Como os dados apresentados eram bem próximos e os valores iniciais dos centroides começavam com valores aleatórios mas baixos não foi apresentada grande diferença entre o k-means e o k-means++ nos resultados. Entretanto, mesmo com essas limitações foi possível observar que o algoritmo de k-means++ possui uma estabilidade e precisão maiores que as do k-means e no geral exige menos iterações para apresentar os resultados.