Taller 5: El algoritmo KMeans

Juan Marín 2422117-3743

Isabela Bermúdez Moreno 2418564-3743

Juan Francisco Díaz

Universidad del Valle
Facultad de ingeniería
Programa académico de ingeniería de sistemas
Noviembre 2025

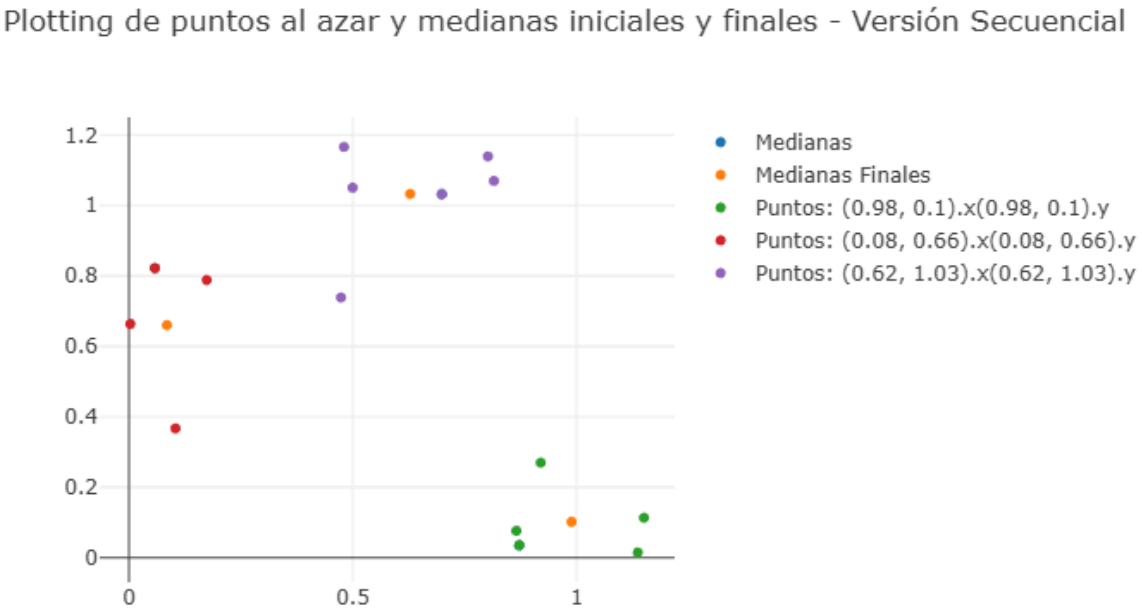
1. Informe corrección

Probaremos que la nuestro implementación del algoritmo kMeans funciona para todos los casos posible, primero veremos que el algoritmo es correcto para algunos casos usando la visualizacion de los resultados de la funcion kMedianasSeq.

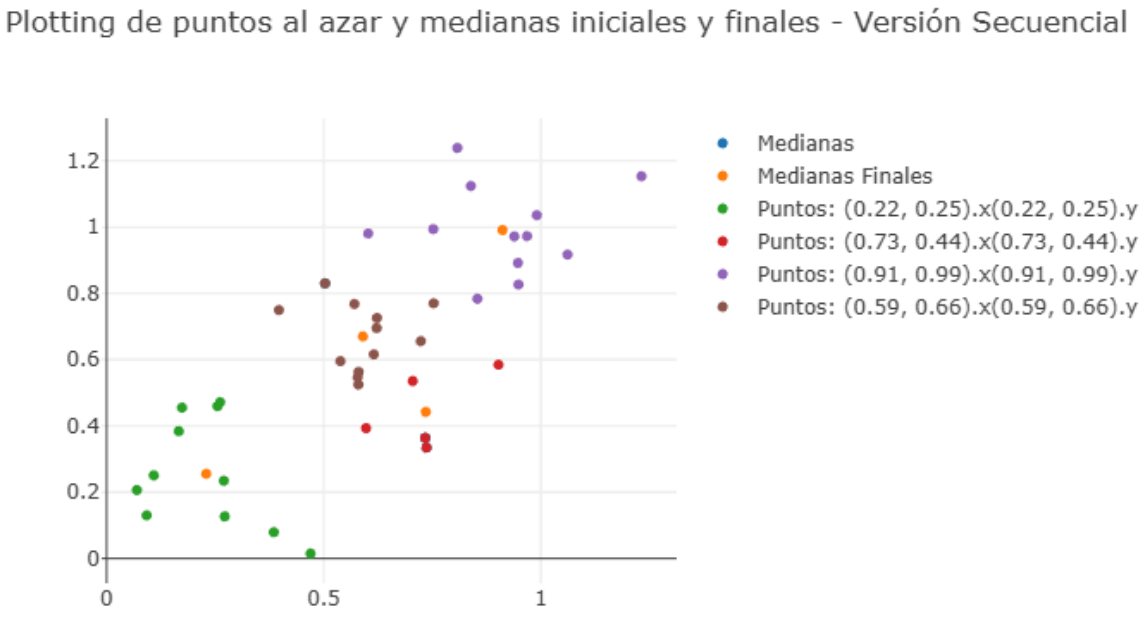
Las gráficas fueron creadas usando un .scala de la siguiente manera pues en el archivo .sc no se exportaban en nuestras máquinas de la siguiente manera:

```scala
import Benchmark._
import kmedianas2D._
import java.nio.file.Paths
import plotly._, element._, layout._

object Pruebas {

  def main(args: Array[String]): Unit = {



    val puntos1 = generarPuntos(4, 40).toSeq
    probarKmedianas(puntos1, 4, 0.01)

    val puntos_2 = generarPuntos(3, 15).toSeq
    probarKmedianas(puntos_2, 3, 0.01)


  }
}

```

1. kMedianas para 15 puntos y 3 clusters



2. kMedianas para 40 puntos y 4 clusters



- **ClasificarSeq.**

Vamos a probar que **clasificarSeq(puntos, medianas)**, donde **puntos** es una secuencia de puntos y **medianas** es una secuencia de puntos, devuelve la clasificación de los puntos según la mediana más cercana, es decir, un mapa

$$C: Map[Punto, Seq[Punto]]$$

tal que las claves son un subconjuntos de las medianas ($keys(C) \subseteq medianas$) , y para toda clave $m \in keys(C)$, el valor $C(m)$ es una secuencia de puntos $Pm \subseteq Puntos$, tal que para cada punto $p \in Pm$, m es la mediana más cercana a p entre todas las medianas en M.

Hipótesis:
Asumimos como hipótesis que la función:

$$hallarPuntoMasCercano(p, medianas)$$

donde p es un punto, y medianas es una secuencias de tipo Punto, devuelve la mediana más cercana al punto p, es decir:

$$hallarPuntoMasCercano(p, medianas) = arg_{\,m \in medianas}\ min\ distanciaAlCuadrado(p, m)$$

Denotemos:

$$M(p) = hallarPuntoMasCercano(p, medianas)$$

Definición funcional
Sea el conjunto de puntos $P = \{p1, p2,..., pn\}$
y el conjunto de medianas $M = \{m1, m2,..., mk\}$

Definamos la función que asigna a cada mediana mi, el conjunto de puntos que le pertenecen, es decir:

$$f(P, M) = \{(mi, \{p \in P: M(p) = mi\}) | mi \in M\}$$

Y sea clasificarSeq el siguiente programa en Scala:

```
def clasificarSeq(puntos: Seq[Punto], medianas: Seq[Punto]): Map[Punto, Seq[Punto]] = {
  puntos.groupBy(p => hallarPuntoMasCercano(p, medianas))
}
```

Demostraremos que:

$$\forall p \in P, \forall m \in M: clasificarSeq(puntos, medianas) = f(puntos, medianas)$$

Usando el método de sustitución.

Caso base:
Cuando puntos = Seq(), se evalua:

$$clasificarSeq(Seq(), medianas) = Seq().groupBy(p => hallarPuntoMasCercano(p, medianas))$$

Por definición de GroupBy en Scala:

$$Seq().groupBy(f) = Map()$$

es decir un mapa vacío.

Esto coincide con la definición de f(P,M), cuando P = ∅ , ya que ningún punto pertenece a ninguna mediana.

$$F(\emptyset, M) = \{\} = Map()$$

Concluimos que:

$$clasificarSeq(Seq(), medianas) = F(\emptyset, M)$$

Caso general: Secuencia no vacía

Sea puntos = Seq(p1,p2,...,pn) con n > 0.

Sustituyendo en la función:

$$clasificarSeq(puntos, medianas) = puntos.groupBy(p \Rightarrow hallarPuntoMasCercano(p, medianas)$$

Por definición de GroupBy tenemos:

$$groupBy(f)(S) = \{(k, \{x \in S: f(x) = k\}) \mid k \in Im(f)\}$$

Donde $Im(f)$ es el conjunto de valores de $f(x)$

Sustituyendo S = puntos y f(x) = f(p) = hallarPuntoMasCercano(p,medianas)=M(p):

tenemos que:

$$clasificarSeq(puntos, medianas) = \{(m, \{p \in puntos: M(p) = m\}) \mid m \in M\}$$

Esto es exactamente la definición de f(P,M).

Podemos concluir que:

$$\forall p \in P\ ,\ \forall m \in M: clasificarSeq(puntos, medianas) = f(puntos, medianas)$$

- **Demostracion equivalencia ClasificarPar**

Queremos demostrar que para todo conjuntos de puntos P y medianas M:

$$clasificarPar(umb)(P,\ M)\ =\ clasificarSeq(P,\ M)$$

donde umb es un entero, representa el umbral por el que se definirá si se usa paralelismo o no.

Hipótesis de inducción:
Asumimos como hipótesis inductiva que:

$$clasificarPar(umb)(Q,\ M)\ =\ clasificarSeq(Q,\ M)$$

Para toda subsecuencia $Q \subseteq P\ tal\ que\ |Q|\ <\ |P|$
Queremos probar que se cumple también para P, tal que $|P|$ = $|Q|$ :: 1

Caso base:
Si $|P|$ < umb, entonces:

$$clasificarPar(umb)(P,\ M)\ =\ clasificarSeq(P,\ M)$$

Caso recursivo:
Si |P| > umb, por sustitución tenemos que:

$$clasificarPar(umb)(P,\ M)\ \rightarrow if\ (P.length\ <=\ umb)\quad clasificarSeq(P,M)\ else\ \{...$$
$$\rightarrow val\ (izq,\ der)\ =\ P.splitAt(p\ .length\ /\ 2)$$
$$val\ (clasifIzq,\ clasifDer)\ =\ parallel(clasificarPar(umb)(izq,\ M),$$
$$clasificarPar(umb)(der,\ M))$$
$$unirClasificaciones(clasifIzq,\ clasifDer)$$

Por la hipótesis inductiva:

$$clasificarPar(umb)(izq,\ M) = clasificarSeq(izq,\ M)$$
$$clasificarPar(umb)(der,\ M)\ =\ clasificarSeq(der,\ M)$$

Tenemos que se clasificaron los puntos izq, correspondientes a la primera mitad de |P| y los puntos der, correspondientes a el resto de |P| .

Luego, clasificarPar une los resultados de la clasificación de izq y der, con la función:

$$unirClasificaciones(clasifIzq,\ clasifDer)\ =\ unirClasificaciones(clasificarSeq(izq,\ M),$$
$$clasificarSeq(der,\ M))$$

Denotemos clasifizq = m1 y clasifDer = m2

Tenemos que unirClasificaciones está definida como:

$$unirClasificaciones(m1, m2)\ \rightarrow (m1.toSeq\ ++\ m2.toSeq).groupBy(\_.\_1).map\ \{\ case\ (mediana,$$

$$listaPares) \Rightarrow mediana \rightarrow listaPares.flatMap(\_.\_2)\}$$

Para cada mediana m ∈ M , la operación toma todas las entradas asociadas a m en ambos mapas parciales (m1 y m2) y concatena las secuencias de puntos asociadas.
Teniendo en cuenta que P = izq :: der
Podemos decir que:

$$unirClasificaciones(m1, m2) = clasificarSeq(izq ++ der, M) = clasificarSeq(P, M)$$

Finalmente podemos concluir que:

$$clasificarPar(umb)(P, M) = clasificarSeq(P, M)$$

- **actualizarSeq**

Queremos probar que la funcion:

actualizarSeq(clasif: Map[Punto, Seq[Punto]], medianasViejas: Seq[Punto]): Seq[Punto]

calcula correctamente el nuevo conjunto de medianas, donde cada nueva mediana corresponde al promedio de los puntos asociados a su mediana anterior en clasif.

Formalmente, si denotamos por:

f:(Map[Punto,Seq[Punto]],Seq[Punto])→Seq[Punto]

tal que:

f(clasif,M)=[promedio(clasif(m1)),promedio(clasif(m2)),…,promedio(clasif(mk))],

donde M=[m1,m2,…,mk], y m_n son las medianas viejas, y clasif denota el Map[Punto, Seq[Punto]] de las medianas viejas y sus puntos. Promedio(S) denota el punto cuyo valor en x e y son los promedios de las coordenadas de los puntos de S asociadas a una mediana m_n ,o solo m en el caso en que S está vacío.

Queremos demostrar que:

∀ clasif,M: actualizarSeq(clasif,M)==f(clasif,M)

Partimos de la definición:

actualizarSeq(clasif,M)=M.map(m⇒calculePromedioSeq(m,clasif.getOrElse(m,Seq())))

Aplicando la semántica de map, sabemos que:

M.map(f) = [f(M(0)),f(M(1)),…,f(M(n−1))]

Entonces, por sustitución:

actualizarSeq(clasif,M)=

[calculePromedioSeq(M(0),clasif.getOrElse(M(0),Seq())),…,calculePromedioSeq(M(n−1),clasif.getOrElse(M(n−1),Seq()))]

La función calculePromedioSeq está definida como:

```
def calculePromedioSeq(medianaVieja: Punto, puntos: Seq[Punto]): Punto =
  if (puntos.isEmpty) medianaVieja
  else new Punto(puntos.map(_.x).sum / puntos.length,
                 puntos.map(_.y).sum / puntos.length)
```

por lo tanto hay dos casos.

1. Si la seq puntos está vacía, se devuelve medianaVieja, que preserva la mediana cuando no hay datos nuevos.
   Esto satisface la condición de estabilidad esperada del algoritmo.

2. Si puntos no es vacío, se devuelve un nuevo Punto con las coordenadas promedio de todos los puntos del grupo,
   que corresponde exactamente a la definición matemática de la nueva mediana.

Así:

calculePromedioSeq(m,clasif(m)) = promedio(clasif(m))

Reemplazando el comportamiento de calculePromedioSeq en la expresión general:

actualizarSeq(clasif,M)=[promedio(clasif(M(0))),…,promedio(clasif(M(n−1)))]

Por definición anterior de f, actualizarSeq es exactamente igual a: f(clasif,M)
Por sustitución y equivalencia funcional:

∀ clasif,M: actualizarSeq(clasif,M)=f(clasif,M)

- **Demostracion equivalencia actualizarPar**

Queremos demostrar que:

$\forall$ clasif:Map[Punto,Seq[Punto]], $\forall$ medianasViejas:Seq[Punto],

actualizarPar(clasif,medianasViejas) = actualizarSeq(clasif,medianasViejas)

Ambas funciones actualizarSeq y actualizarPar tienen la **misma estructura funcional**:
usan map sobre la secuencia medianasViejas, aplicando una transformación pura a cada elemento:

Sea:

f(mediana) = calculePromedioX(mediana,clasif.getOrElse(mediana,Seq()))

La única diferencia reside en si la función auxiliar calculePromedioX usa estructuras **secuenciales** o **paralelas** para el cálculo del promedio.

función promedio par:

```
def calculePromedioPar(medianaVieja: Punto, puntos: Seq[Punto]): Punto = {
   if (puntos.isEmpty) medianaVieja
   else {
     val puntosPar = puntos.par
     new Punto(
       puntosPar.map(p => p.x).sum / puntos.length,
       puntosPar.map(p => p.y).sum / puntos.length
     )
   }
 }
```

Para cualquier puntos: Seq[Punto], se cumple:

Si puntos.isEmpty, ambos devuelven medianaVieja.
→ igualdad trivial.

Si puntos.nonEmpty, ambos calculan:
nuevo x= promedio de las coordenadas x de cada punto
nuevo y= promedio de las coordenadas y de cada punto

La diferencia entre map y puntos.par.map es **solo operacional**, ambas producen la **misma secuencia de resultados** en términos de contenido, aunque una lo haga en paralelo.

$$\sum(puntos.map(f)) \quad = \quad \sum(puntos.par.map(f))$$

Se demuestra que:

$\forall$ clasif, $\forall$ medianasViejas.

actualizarPar(clasif,medianasViejas) = actualizarSeq(clasif,medianasViejas)

- **hayConvergenciaSeq**

La función evalúa si todas las medianas nuevas (mN) están suficientemente cerca de las medianas viejas (mV), de acuerdo con un umbral eta (α).

sea: $f(\alpha,mV,mN)=\{$ true si $\forall\, i < |mV|$, distanciaAlCuadrado(mVi,mNi)$\leq\alpha^2$; false en otro caso

Demostraremos que la función hayConvergenciaSeq devuelve **true** si todas las distancias cuadradas son menores o iguales a α^2, y **false** si alguna supera dicho valor.

formalmente decimos queremos demostrar por estado e invariante que:

hayConvergenciaSeq(α, mV, mN) = f(α, mV, mN)

- Estado:
  s=(mV,mN) donde mV y mN son secuencias de puntos de igual longitud.

- Estado inicial:
  s0=(medianasViejas,medianasNuevas) las listas completas que se reciben al inicio.

- Estado final:
  sf se alcanza cuando mViejas.isEmpty, es decir, cuando ya se han comparado todos los pares de medianas y se haya cumplido la condición o la condición haya sido false para un par de medianas .

- Transformación:
  transformar(s) = (mV.tail,mN.tail), en cada paso se eliminan las cabeceras de ambas secuencias.

- Invariante:
  Inv(s)≡ “Todas las medianas procesadas hasta el momento cumplen con: distanciaAlCuadrado(mV ,mN) ≤ α^2 .

Inv(s0)

Al inicio no se ha procesado ningún par de medianas, por lo tanto, la condición “todas las comparaciones hechas hasta ahora cumplen el criterio” es verdadera.
⇒ Inv(s0) se cumple.

Preservación del invariante

Sea = s_i = (mV, mN) un estado no final.
Si Inv(s_i) es verdadero y la distancia entre las cabeceras mV.head y mN.head cumple:

$distancia^2 \leq \alpha^2$
Entonces al transformar:

transformar(s_i)=(mV.tail,mN.tail)

las comparaciones anteriores siguen siendo válidas y se agrega una nueva que también cumple la condición.

concluimos que se cumple:

⇒ Inv(transformar(s_i))

Corrección en el estado final
Cuando se alcanza el estado final sf=(List(),List()), la Inv(sf) garantiza que todas las comparaciones realizadas cumplieron el criterio de convergencia.

La función devuelve true, lo que coincide con la especificación:
“Todas las medianas nuevas están dentro del radio de convergencia respecto a las viejas”.

⇒ Inv(sf) → respuesta(sf) = true

Terminación

En cada paso, las secuencias mV y mN disminuyen en un elemento. Por lo tanto, después de un número finito de pasos igual a su longitud inicial, se alcanza el estado final List() por lo que el algoritmo termina.

Por los cuatro puntos anteriores, se cumple:

∀(mV,mN):

hayConvergenciaSeq(eta,Viejas,mN) = f(eta,Viejas,mN)

- **Demostración equivalencia hayConvergenciaPar**

Se desea demostrar que las versiones secuencial y paralela del método hayConvergencia son algorítmicamente equivalentes,
es decir, que para cualquier entrada producen el mismo resultado lógico.

Demostraremos formalmente que:

∀ (α, mV, mN): hayConvergenciaPar(α, mV, mN) = hayConvergenciaSeq(α, mV, mN)

Ambas funciones verifican la misma propiedad: que todas las medianas nuevas estén dentro del umbral de convergencia α respecto a las medianas viejas, la diferencia está únicamente en la estrategia de evaluación:

- hayConvergenciaSeq evalúa los pares (mVi, mNi) secuencialmente.

- hayConvergenciaPar divide la secuencia en subproblemas y los evalúa en paralelo.

El resultado no cambia porque la operación lógica de conjunción, (&&), es conmutativa y asociativa.

Definimos una función como antes f(α, mV, mN) que ambas implementan:

$$\text{sea: } f(\alpha,mV,mN)=\begin{cases} \text{true} & \text{si } \forall\, i < |mV|,\ \text{distanciaAlCuadrado}(mVi,mNi)\leq\alpha^2 \\ \text{false} & \text{en otro caso} \end{cases}$$

Caso base (mV.length ≤ umb):
La versión paralela evalúa directamente la condición sobre todos los pares mediante zip y forall, equivalente a recorrer secuencialmente los pares, ambas funciones realizan la misma operación de verificación directa, por lo tanto son equivalentes.

Caso recursivo (mV.length > umb):
Divide las secuencias en dos mitades independientes y las evalúa en paralelo:

```
(val resIzq, val resDer) = parallel(
        hayConvergenciaPar(η, mV_izq, mN_izq),
        hayConvergenciaPar(η, mV_der, mN_der)
)
resIzq && resDer
```

Suponemos que la equivalencia se cumple para todas las listas de tamaño menor que n.
Para una lista de longitud n, la versión paralela lo divide en dos subproblemas y combina con, &&. Por hipótesis inductiva, cada subproblema produce el mismo resultado que su versión secuencial, y la conjunción final también es equivalente al cálculo secuencial completo.

Por tanto, por inducción estructural sobre el tamaño de la lista, se cumple que:

$$\text{hayConvergenciaPar}(\alpha, mV, mN) = \text{hayConvergenciaSeq}(\alpha, mV, mN)$$

- **KmedianasSeq**

Sea

$$f: Seq[Punto] \times Seq[Punto] \times R \rightarrow Seq[Punto]$$

la función que recibe:
una secuencia de puntos **p** de tipo Seq[Puntos],que representa todos los puntos por clasificar
una secuencia de medianas m de tipo Seq[Puntos]
y un valor fijo eta de tipo Double

Y devuelve una nueva secuencia de medianas m' de tipo Seq[Puntos] resultado de clasificar los puntos p con respecto a la medianas m y actualizar cada mediana con el promedio de los puntos que le corresponden. El proceso finaliza cuando la distancia al cuadrado entre las medianas viajas y las nuevas es menor o igual a eta^2.

Y sea Pf el siguiente programa en Scala:

```
@tailrec
final def kMedianasSeq(puntos: Seq[Punto], medianas: Seq[Punto], eta: Double): Seq[Punto] = {

  val clasif: Map[Punto, Seq[Punto]] = clasificarSeq(puntos, medianas)

  val nuevasMedianas: Seq[Punto] = actualizarSeq(clasif, medianas)

  val convergieron = hayConvergenciaSeq(eta, medianas, nuevasMedianas)

  if (convergieron)
    nuevasMedianas
  else
    kMedianasSeq(puntos, nuevasMedianas, eta)
}
```

Vamos a argumentar sobre la corrección del programa Pf usando el método de estado invariante.

Vamos a demostrar que:

$$\forall n \in N, \forall eta \in R: Pf(Seq(a1, a2, ..., an), Seq(b1, b2, ..., bn), eta) = f(Seq(a1, a2, ..., an), Seq(b1, b2, ..., bn), eta)$$

Definición de estados: Un estado de la iteración se representa como:

$$s = (p, m, eta, t)$$

donde:

- p representa la secuencia de puntos a clasificar (tipo Seq[Punto]) , es constante durante todo el proceso.
- m es la secuencia actual de medianas o clusters (tipo Seq[Punto]).
- eta es el umbral de convergencia (tipo Double), fijo durante la ejecución.
- t es el número de iteraciones (t no está explícito en el código, pero lo consideramos para razonar formalmente).

Estado inicial s_0:

$$s_0 = (p, m_0, eta, 0)$$

donde **p** es la secuencia de puntos**,** m_0 la secuencia de medianas iniciales, **eta** el umbral de convergencia.

Estado final s_f: Un estado s = (p,m,eta,t) es final cuando:

$$hayConvergenciaSeq(eta, m, m') = true$$

donde **m'** es la colección de medianas calculadas a partir de **m** en la iteración actual.

El estado es final cuando se cumple que:

$$\forall i: distancialAlCuadrado(m_i, m'_i) \leq eta^2$$

Transformación:

transformar(p,m,eta,t) = (p,m',eta,t+1)

donde m' es el resultado de:

1. clasif = clasificarSeq(p,m) (asignar a cada punto su mediana más cercana)
2. m' = actualizarSeq(clasif, m) (calcular promedios a cada mediana o cluster)

Invariante:

$Inv(s) \equiv$ "m es la colección de medianas producida tras aplicar t iteraciones sobre $m0$."

Corrección:

1. $Inv(s_0)$

   El estado inicial $s_0 = (p, m_0, eta, 0)$ cumple con el invariante, ya que m_0 corresponde a las medianas iniciales. (0 iteraciones realizadas).

2. **Preservación del invariante:**

   Queremos probar que:

   $$(s_i \neq s_f \wedge Inv(s_i)) \Rightarrow Inv(transformar(s_i))$$

   Supongamos un estado $s_i = (p, m, eta, t)$ no final.

Si $Inv(s_i)$ es verdadero, entonces m corresponde a la colección de medianas obtenidas tras aplicar t iteraciones sobre m0, las medianas iniciales.

Entonces al transformar:

$$transformas(s_i) = (p, m', eta, t + 1)$$

Donde m' = $hayConvergenciaSeq(eta, m, m') = true$.

Por definición de transformación, m' corresponde a las medianas obtenidas tras aplicar t + 1 iteraciones sobre m_0**.** Por tanto el invariante también se cumple.

concluimos:

$$(s_i \neq s_f \wedge Inv(s_i)) \Rightarrow Inv(transformar(s_i))$$

3. Corrección en el estado final

Cuando se alcanza el estado final, se cumple que:

$$hayConvergenciaSeq(eta, m, m') = true.$$

Entonces las medianas nuevas m', calculadas a partir de las medianas viejas m, satisfacen:

$$\forall i: distancialAlCuadrado(m_i, m'_i) \leq eta^2$$

$Inv(s_f)$ garantiza que las medianas nuevas m' sean válidas.

Por lo tanto la salida m' satisface el criterio de convergencia y el programa devuelve m' como resultado correcto.

4. Terminación:

En cada iteración, el programa genera una nueva colección de medianas o clusters m' a partir de las medianas anteriores m.

Dado que las medianas se recalculan promediando los puntos asignados a las medianas anteriores, al diferencia $distancialAlCuadrado(m_i, m'_i)$ tiende a disminuir a medida que el proceso se repite.

Por el criterio de parada $hayConvergenciaSeq(eta, m, m')$, el algoritmo finaliza cuando todas las medianas cambian menos que el umbral eta, es decir:

$$\forall i: distancialAlCuadrado(m_i, m'_i) \leq eta^2$$

En ese momento el programa retorna m'.

Conclusión:

Demostrados los puntos anteriores, se puede concluir que:

$\forall n \in N, \forall eta \in R: Pf(Seq(a1, a2, \ldots, an), Seq(b1, b2, \ldots, bn), eta) = f(Seq(a1, a2, \ldots, an), Seq(b1, b2, \ldots, bn), eta)$

- **Demostracion equivalencia kmedianasPar:**

Queremos demostrar que:

Sea:

- M = medianas:Seq[Punto],
- P = puntos:Seq[Punto]
- eta:Double

$$\forall P, M, eta: kMedianasSeq(P, M, eta) = kMedianasPar(P, M, eta)$$

Ambas funciones, kmedianasSeq y kmedianasPar siguen la misma estructura recursiva de cola, en cada iteración:

1. Clasifican los puntos según su mediana más cercana.
2. Calculan nuevas medianas promediando los puntos de cada grupo.
3. Verifican si las medianas nuevas cambian menos que el umbral eta respecto a las medianas anteriores:
    - si se cumple esta condición, el proceso finaliza.
    - Si no hay convergencia, se realiza la llamada recursiva con las medianas nuevas.

La única diferencia es que kmedianasPar distribuye el trabajo en varios threads, logrando la paralización de las tareas.

Para esto, la funcion kmedianasPar usa versiones paralelas de mismas funciones que se emplean en kmedianasSeq.

Como ya se demostró anteriormente, las siguientes funciones son equivalentes funcionalmente a sus versiones secuenciales:

$$\forall p, m: clasificarPar(umb)(p, m) = clasificarSeq(p, m).$$

$donde\ p: Seq[Puntos],\ m: Seq[Puntos]\ y\ umb: double$

$$\forall clasif, m: actualizarPar(clasif, m) = actualizarSeq(clasif, m)$$

$donde\ clasif: Map[Punto,\ Seq[Punto]],\ m: Seq[Punto].$

$$\forall m, m', \eta: hayConvergenciaPar(\eta, m, m') = hayConvergenciaSeq(\eta, m, m')$$

$Sea\ m: Seq[Puntos], m': Seq[Puntos]\ y\ eta: Double$

Estas igualdades garantizan que ambas implementaciones aplican las mismas transformaciones a los mismos datos en cada iteración.

Además, en ambos programas, en cada llamada recursiva se cumple que:

- Devuelven m' (las medianas nuevas) si:

$$hayConvergenciaSeq(eta, m, m') = true.$$

- Llama recursivamente a kmedianasPar(p,m',eta) o kmedianasSeq(p,m',eta) en caso contrario.

Por tanto, podemos concluir que:

$$\forall P, M, eta: kMedianasSeq(P, M, eta) = kMedianasPar(P, M, eta)$$

informe desempeño

Para realizar la comparación del desempeño sobre la implementación del algoritmo k-means en su versión secuencial y paralela se usó el método tiemposKmedianas(puntos:Seq[Punto], k:Int, eta:Double), que recibe los puntos a dividir en k clusters, con un umbral de convergencia eta de 0.01, y devuelve una tupla con los tiempos del algoritmo secuencial, paralelo, y la aceleración del algoritmo paralelo con respecto al secuencial.

Los ejemplos de prueba se generaron utilizando la función: generarPuntos(k,numPuntos), donde k es el número de clusters y numPuntos es la cantidad de puntos a genera, esta función nos da los puntos para agrupar en k clusters.

Se realizaron las pruebas variando dos parámetros:

El **número de clústeres** (*k*): 2, 4, 8, 16 y 32.
El **número de puntos** generados: 512, 2048, 8192, 32768 y 131072.

Cada combinación de valores se ejecutó tanto en la versión secuencial como en la versión paralela del algoritmo.

(k, puntos, 0.01)	Tiempo k-medianas secuencial	Tiempo k-medianas paralelo	Aceleración
(2, 512)	1.3183 ms	24.4699 ms	0.0539
(2, 2048)	3.8473 ms	6.8435 ms	0.5622
(2, 8192)	13.2345 ms	16.3451 ms	0.8097
(2, 32768)	52.0568 ms	54.8852 ms	0.9485
(2, 131072)	271.6978 ms	200.1494 ms	1.3575
(4, 512)	8.6661 ms	40.5968 ms	0.2135
(4, 2048)	17.8541 ms	50.3359 ms	0.3547
(4, 8192)	167.8648 ms	119.5256 ms	1.4044
(4, 32768)	391.7772 ms	272.6461 ms	1.4369
(4, 131072)	1709.4412 ms	1141.528 ms	1.4975
(8, 512)	5.154 ms	27.4038 ms	0.1881
(8, 2048)	10.2301 ms	22.6725 ms	0.4512
(8, 8192)	131.1009 ms	134.9315 ms	0.9716
(8, 32768)	333.0609 ms	275.3425 ms	1.2096
(8, 131072)	1898.0949 ms	1250.7122 ms	1.5176
(16, 512)	12.4705 ms	38.2061 ms	0.3264
(16, 2048)	41.5592 ms	34.6773 ms	1.1985
(16, 8192)	131.1044 ms	100.6182 ms	1.303
(16, 32768)	805.6837 ms	502.1452 ms	1.6045
(16, 131072)	2267.3468 ms	1087.5772 ms	2.0848
(32, 512)	34.698 ms	67.5783 ms	0.5134
(32, 2048)	162.7869 ms	155.2405 ms	1.0486
(32, 8192)	414.8405 ms	228.5059 ms	1.8154
(32, 32768)	1215.7193 ms	514.2167 ms	2.3642
(32, 131072)	5391.2123 ms	1897.7947 ms	2.8408
(64, 512)	51.2118 ms	116.2116 ms	0.4407
(64, 2048)	192.0582 ms	172.4784 ms	1.1135
(64, 8192)	1401.6602 ms	707.6382 ms	1.9808
(64, 32768)	4255.509 ms	1323.4506 ms	3.2155
(64, 131072)	29037.9257 ms	9194.4304 ms	3.1582
(128, 512)	183.383 ms	232.7762 ms	0.7878
(128, 2048)	603.0809 ms	468.3868 ms	1.2876
(128, 8192)	2351.3501 ms	644.5631 ms	3.648

(128, 32768)	6058.8822 ms	3376.787 ms	1.7943
(128, 131072)	48661.4778 ms	12442.3562 ms	3.911

Análisis de los resultados

La versión paralela muestra una mejora progresiva en el rendimiento conforme aumenta el tamaño de los datos y el número de clústeres. En los escenarios de menor escala (512–2048 puntos), la sobrecarga de coordinación del paralelismo —asociada a la creación y sincronización de tareas— supera el beneficio obtenido por la ejecución concurrente, resultando en tiempos de ejecución mayores que en la versión secuencial y aceleraciones menores a 1.

A partir de tamaños de entrada medios y grandes (≥ 8192 puntos), la aceleración supera el valor de 1, lo que evidencia un uso más eficiente de los núcleos del procesador. En estos casos, el paralelismo logra distribuir la carga de trabajo de forma efectiva, reduciendo significativamente el tiempo total de ejecución frente a la versión secuencial.

Para los tamaños de datos más grandes (por encima de 100 000 puntos), la versión paralela alcanza aceleraciones cercanas a 3×, mostrando que el algoritmo escala adecuadamente con el volumen de datos y aprovecha de manera óptima la arquitectura multinúcleo.

En conclusión, los resultados confirman la correcta implementación y el beneficio del paralelismo de tareas y de datos dentro del algoritmo *k-Medianas*. Si bien la versión secuencial es más eficiente en problemas pequeños, la versión paralela demuestra su ventaja en escenarios de alta demanda computacional, mejorando la eficiencia global y reduciendo los tiempos de cómputo de forma significativa.

Análisis comparativo de las diferentes soluciones

El algoritmo de *k-medianas* se implementó en dos versiones: una secuencial y una paralela. El objetivo fue comparar el desempeño entre ambas para evaluar los efectos del

paralelismo de datos y de tareas. Las pruebas se realizaron con distintos tamaños de entrada y número de clústeres, midiendo el tiempo de ejecución de cada versión y calculando la aceleración, definida como la razón entre el tiempo secuencial y el tiempo paralelo. Si la aceleración es mayor que uno, significa que la versión paralela fue más eficiente.

En la versión paralela se aplicaron dos tipos de paralelismo. El primero fue el **paralelismo de datos**, presente en las funciones calculePromedioPar y actualizarPar, donde se usó la estructura par.map para distribuir las operaciones sobre los puntos y las medianas entre varios núcleos del procesador. Esto permite procesar subconjuntos de datos en simultáneo, ideal cuando el número de puntos o medianas es grande.

El segundo tipo fue el **paralelismo de tareas**, implementado en las funciones clasificarPar y hayConvergenciaPar. En este caso, se usa la construcción parallel(expr1, expr2) para dividir el problema en dos subproblemas independientes que se ejecutan al mismo tiempo. Esta estrategia aprovecha el hecho de que las particiones del conjunto de puntos pueden clasificarse y evaluarse de forma autónoma. Finalmente, el algoritmo kMedianasPar combina ambos enfoques: aplica paralelismo de tareas a nivel de control del flujo y paralelismo de datos en las operaciones internas, lo que permite aprovechar mejor los recursos disponibles.

Los resultados experimentales muestran que para entradas pequeñas la versión secuencial es más rápida. En esos casos, la sobrecarga asociada a crear y coordinar hilos paralelos supera cualquier posible ganancia, lo que se refleja en aceleraciones menores que 1. Sin embargo, a medida que aumenta el tamaño de los datos y el número de clústeres, la versión paralela comienza a superar claramente a la secuencial.

En los primeros experimentos (por ejemplo, con 3 o 5 clústeres y pocos puntos), los tiempos paralelos fueron ligeramente mayores. Pero a partir de entradas medianas (r9–r15) ya se observa una aceleración por encima de 1.3, y en los conjuntos grandes (r39 en adelante) el rendimiento mejora significativamente, alcanzando aceleraciones cercanas a 4. Esto demuestra que el paralelismo empieza a ser realmente beneficioso cuando la cantidad de datos es suficiente para amortiguar la sobrecarga del procesamiento concurrente.

El paralelismo de datos mostró ser más eficiente en las fases donde hay operaciones homogéneas y fácilmente distribuibles, como los cálculos de promedios o sumas sobre los puntos. En cambio, el paralelismo de tareas funciona mejor cuando el problema puede dividirse en subtareas grandes e independientes, como las que se resuelven en clasificarPar y hayConvergenciaPar. En estos casos, cada tarea puede ejecutarse en paralelo sin necesidad de comunicación entre hilos hasta la etapa final de combinación de resultados.

Combinando ambos enfoques se logra un equilibrio entre granularidad y aprovechamiento de los núcleos del procesador. El resultado es que el algoritmo paralelo mantiene un comportamiento similar al secuencial en entradas pequeñas, pero obtiene aceleraciones crecientes y sostenidas conforme aumentan el tamaño y complejidad del problema.

En conclusión, las paralelizaciones implementadas sí resultaron efectivas. La versión paralela conserva la misma lógica algorítmica que la secuencial, pero distribuye el trabajo

de manera más eficiente en problemas grandes. El paralelismo de tareas es más útil cuando el trabajo puede separarse naturalmente en partes independientes, mientras que el de datos es más ventajoso cuando se realizan operaciones repetitivas sobre grandes volúmenes. La combinación de ambos, como se hizo en esta implementación, permite escalar adecuadamente el rendimiento sin modificar la corrección del algoritmo.