Documentació d'Estructures de Dades i Algorismes

Grup 14 - Subgrup 4
Pol Gay Hérnandez (pol.gay)
Mateus Grandolfi Albuquerque (mateus.grandolfi)
Wenqiang He (wenqiang.he)
Fardin Arafat Mia Akter (fardin.arafat.mia)

Lliurament 2.0 - Tardor 2025-26

Índex

1 Estructures de Dades Clau del Projecte 3

1.1 Package `user` . . . 3
1.2 Package `Survey` . . . 3
1.3 Package `Response` . . . 4
1.4 Package `Encoder` . . . 5
1.5 Package `app.controller` . . . 5
1.6 Package `kmeans` i `kselector` . . . 5
1.7 Capa de Persistència (`persistence`) . . . 6
1.8 Capa de Presentació (`domain.app`) . . . 7

2 Visió General del Procés d'Anàlisi 8

3 Algorisme de Codificació (OneHotEncoder) 8

4 Algorisme de Clustering K-Means 10

4.1 Càlcul de Representants . . . 11

5 Algorisme d'Inicialització K-Means++ 12

6 Algorisme de Validació (Silhouette) 13

7 Algorisme de Selecció de K (Mètode del Colze) 15

A Annex: Pseudocodis dels Algorismes 17

A.1 Pseudocodi de OneHotEncoder.transform (actualitzat) . . . 17
A.2 Pseudocodi de l'Algorisme K-Means (Lloyd) . . . 18
A.3 Pseudocodi de la Inicialització K-Means++ . . . 19
A.4 Pseudocodi del Coeficient de Silhouette (per punt) . . . 20
A.5 Pseudocodi del Mètode del Colze . . . 21

1 Estructures de Dades Clau del Projecte

Per implementar la lògica del projecte, s'han seleccionat estructures de dades centrant-nos en l'eficiència de les operacions més comunes.

1.1 Package `user`

- **Classe:** `AuthService`
  - **Estructura:** `Map<String, RegisteredUser> registeredUsers`
    - **Implementació:** `HashMap`
    - **Cost:** Temps mitjà de $O(1)$ per a `put` (registre) i `get` (cerca per ID). Temps en el pitjor cas de $O(n)$.
    - **Justificació:** S'utilitza un `HashMap` per emmagatzemar els usuaris registrats, on la clau és l'ID d'usuari. El `HashMap` calcula un índex (un hash) a partir de la clau. Això permet un accés directe a la memòria on es troba l'objecte, sense haver de recórrer altres elements. Per tant, les operacions de cerca (necessàries pel `login`) i inserció (`register`) tenen un cost mitjà constant. El pitjor cas, $O(n)$, només ocorreria en cas de col·lisions massives de hash, un escenari molt poc probable.
  - **Estructura:** `Map<String, Sesion> activeSessions`
    - **Implementació:** `HashMap`
    - **Cost:** Temps mitjà de $O(1)$ per a `put` (login) i `remove` (logout).
    - **Justificació:** La justificació és idèntica a l'anterior. Les sessions actives es gestionen amb un `HashMap` on la clau és l'ID de la sessió, permetent validar, actualitzar i eliminar sessions de manera molt eficient ($O(1)$ en temps mitjà).

1.2 Package `Survey`

- **Classe:** `Survey`
  - **Estructura:** `List<Question> questions`
    - **Implementació:** `ArrayList`
    - **Cost:** `add` (afegir al final) és $O(1)$ amortitzat. `get(index)` és $O(1)$. `remove(element)` és $O(n)$.
    - **Justificació:** Una enquesta ha de mantenir l'ordre en què es van afegir les preguntes. Un `ArrayList` garanteix aquest ordre d'inserció i proporciona un accés ràpid per índex (posició). L'`ArrayList` utilitza un array intern; afegir un element al final (`add`) té un cost de $O(1)$ amortitzat (la majoria de vegades és $O(1)$, però si l'array intern està ple, s'ha de redimensionar, una operació $O(n)$, tot i que això passa poques vegades). L'accés per posició (`get(index)`) és $O(1)$ perquè es pot calcular directament l'adreça de memòria. L'eliminació (`deleteQuestion`) té un cost $O(n)$, ja que pot requerir desplaçar tots els elements posteriors.
- **Classes:** `SingleChoiceQuestion` i `MultipleChoiceQuestion`
  - **Estructura:** `List<ChoiceOption> options`
    - **Implementació:** `ArrayList`
    - **Cost:** `add` (afegir al final) és $O(1)$ amortitzat.

    * **Justificació:** De la mateixa manera que les preguntes, les opcions s'han de mostrar a l'usuari en l'ordre definit durant la creació. L'ArrayList és la solució estàndard per mantenir una col·lecció ordenada amb un cost d'addició $O(1)$ amortitzat.

- **Classe:** LocalPersistence
  - **Estructura:** Map<String, Survey> surveys
    * **Implementació:** HashMap
    * **Cost:** Temps mitjà de $O(1)$ per a put, get i remove.
    * **Justificació:** Aquesta classe actua com una base de dades en memòria. L'ús d'un HashMap és fonamental per simular l'accés eficient a dades (enquestes) mitjançant una clau primària (l'ID). Permet operacions loadSurvey, saveSurvey i removeSurvey amb una complexitat mitjana de $O(1)$ gràcies al càlcul del *hash* de la clau.
  - **Estructura:** Map<String, SurveyResponse> responses
    * **Implementació:** HashMap
    * **Cost:** Temps mitjà de $O(1)$ per a put, get i remove.
    * **Justificació:** Idèntic al cas de surveys, s'utilitza un HashMap per a les respostes per garantir un accés i modificació en temps constant ($O(1)$) basat en l'ID de la resposta.

1.3 Package Response

- **Classe:** SurveyResponse
  - **Estructura:** List<Answer> answers
    * **Implementació:** ArrayList
    * **Cost:** add és $O(1)$ amortitzat.
    * **Justificació:** Una SurveyResponse agrupa el conjunt de respostes d'un usuari. S'utilitza un ArrayList per la seva simplicitat i eficiència a l'hora d'afegir noves respostes (addAnswer) durant el procés de contestació de l'enquesta, ja que afegir al final té un cost $O(1)$ amortitzat.
- **Classe:** MultipleChoiceAnswer
  - **Estructura:** List<Integer> optionIds
    * **Implementació:** ArrayList (internament), exposat com a Collections.unmodifiableList
    * **Cost:** Creació (còpia) és $O(n)$, on n és el nombre d'opcions triades.
    * **Justificació:** En el constructor, es rep una col·lecció i es crea una nova instància d'ArrayList. Aquesta còpia té un cost de $O(n)$, on n és el nombre d'opcions seleccionades. Com que la llista es fa immutable (només lectura) després de la creació, els costos d'accés (iteració) són lineals ($O(n)$) i eficients per a un nombre petit d'opcions.

1.4 Package `Encoder`

- **Classe:** `OneHotEncoder`
  - **Estructures:** `List<String> featureNames` i `List<Question> orderedQuestions`
    - **Implementació:** `ArrayList`
    - **Cost:** `add` (al final) és $O(1)$ amortitzat. `get(index)` és $O(1)$.
    - **Justificació:** L'ordre és crític. L'accés per índex ha de ser $O(1)$ per garantir una correspondència ràpida entre l'índex de la llista i la columna de la matriu numèrica de sortida. L'`ArrayList` ho compleix. L'addició durant la fase de `fit` és $O(1)$ amortitzat.
  - **Estructures:** `Map<Integer, Map<String, Integer>> textVocab` i `categoricalVocab`
    - **Implementació:** `HashMap`
    - **Cost:** `put` (en `fit`) i `get` (en `transform`) són $O(1)$ en mitjana.
    - **Justificació:** Durant la transformació, necessitem accés ràpid per mapar paraules (String) a índexs (Integer) per l'estratègia Bag of Words, així com mapar IDs de preguntes. El `HashMap` proporciona aquesta cerca eficient.

1.5 Package `app.controller`

- **Classe:** `AnalyticsController`
  - **Estructura:** `Map<Integer, Long> counts`
    - **Implementació:** `LinkedHashMap`
    - **Cost:** `merge` (equivalent a `get` + `put`) és $O(1)$ en mitjana. L'iteració és $O(k)$ on k és el nombre de clústers.
    - **Justificació:** S'utilitza per al recompte de respostes per clúster. Es tria `LinkedHashMap` específicament (en lloc de `HashMap`) perquè manté l'ordre d'inserció. El cost d'actualitzar el comptador (`merge`) és $O(1)$ de mitjana, igual que un `HashMap`. La diferència clau és que `LinkedHashMap` manté addicionalment una llista doblement enllaçada interna que registra l'ordre d'inserció. Això permet que, en iterar sobre el mapa per mostrar els resultats, els clústers apareguin sempre ordenats (Clúster 0, Clúster 1, Clúster 2...), la qual cosa té un petit sobrecost en memòria però millora la presentació a l'usuari.

1.6 Package `kmeans` i `kselector`

- **Classes:** `KMeans` i `ElbowMethod`
  - **Estructura:** `List<Integer>` o `List<Double>`
    - **Implementació:** `ArrayList`
    - **Cost:** `add` és $O(1)$ amortitzat. `Collections.shuffle` (a `KMeans`) és $O(n)$.
    - **Justificació:** Aquestes classes utilitzen `ArrayList` com a estructura de dades temporal durant l'execució dels algorismes. A `KMeans`, s'utilitza per contenir els índexs dels punts de dades. L'operació `Collections.shuffle`, que s'executa un cop a la inicialització, té un cost de $O(n)$ per reordenar els elements. A `ElbowMethod`, s'utilitza per emmagatzemar la llista de puntuacions d'inèrcia per a cada k provat; l'operació `add` al final de la llista és $O(1)$ amortitzat.

- **Classe:** `ClusterModel`
  - **Estructura:** `double[][] centroids`
    - **Implementació:** Matriu primitiva de Java (Array d'Arrays).
    - **Cost:** Accés per índex $O(1)$.
    - **Justificació:** Per emmagatzemar els centroides resultants, utilitzem arrays primitius en lloc de llistes d'objectes per maximitzar l'eficiència numèrica i de memòria. Aquesta estructura s'accedeix milers de vegades dins dels bucles matemàtics de l'algorisme K-Means, per la qual cosa evitem la sobrecàrrega (overhead) de les classes **wrapper** com `Double` o `ArrayList`.
  - **Estructura:** `int[] labels`
    - **Implementació:** Array primitiu.
    - **Cost:** Accés per índex $O(1)$.
    - **Justificació:** Aquest array emmagatzema l'assignació de clúster per a cada punt de dades. L'índex de l'array correspon a l'índex de l'usuari/resposta, permetent un accés directe $O(1)$ per consultar a quin grup pertany qualsevol individu.

1.7 Capa de Persistència (`persistence`)

En aquesta capa, l'objectiu és optimitzar les operacions d'entrada/sortida (I/O) i la manipulació de cadenes per a la serialització manual.

- **Classe:** `UserPersistence` (i Serialitzadors)
  - **Estructura:** `StringBuilder`
    - **Implementació:** `java.lang.StringBuilder` (Array dinàmic de caràcters mutable).
    - **Cost:** $O(1)$ amortitzat per a `append` i $O(N)$ per a `toString`.
    - **Justificació:** A `UserPersistence`, la serialització a format JSON es realitza manualment concatenant camps. Si utilitzéssim la classe `String` i l'operador `+`, estaríem creant una nova instància de String a cada concatenació, resultant en un cost quadràtic $O(N^2)$ inacceptable per a fitxers grans. `StringBuilder` permet construir la cadena final modificant un array intern, garantint un cost lineal $O(N)$.
- **Classes:** `SurveyPersistence`, `ResponsePersistance`
  - **Estructura:** `List<T> buffer`
    - **Implementació:** `ArrayList`
    - **Cost:** $O(1)$ amortitzat per a `add`.
    - **Justificació:** Actua com a *buffer* de memòria intermèdia durant les operacions de lectura massiva (`loadAll`). Els fitxers es llegeixen línia a línia o bloc a bloc, es transformen en objectes i s'acumulen en aquesta llista abans de retornar el control a la capa de domini.

1.8 Capa de Presentació (`domain.app`)

Aquesta capa no només captura dades, sinó que necessita estructures auxiliars per presentar la informació de manera eficient i creuar dades (IDs vs Noms) abans de mostrar-les a l'usuari.

- **Classe:** `TerminalDriver`
  - **Estructura:** `Map<String, String> displayCache` (mètodes com `viewRegisteredResponses`)
    - **Implementació:** `HashMap`
    - **Cost:** $O(1)$ per a `get` i `put`.
    - **Justificació:** Per mostrar llistes de respostes on apareix l'ID d'usuari i l'ID d'enquesta, necessitem mostrar els seus noms reals (DisplayName / Title). En lloc de fer una cerca lineal a la llista d'usuaris per a cada resposta (el que resultaria en un cost $O(N \cdot M)$), pre-carreguem els noms en un `HashMap`. Això permet fer un "join"en memòria amb cost $O(1)$ per a cada element a mostrar.
  - **Estructura:** `List<Survey> selectionMenu`
    - **Implementació:** `ArrayList`
    - **Cost:** $O(1)$ per accés posicional (`get`).
    - **Justificació:** Els controladors retornen `Collection` (sense ordre garantit ni accés per índex). Per generar menús numèrics ("Seleccioni l'opció 1, 2..."), bolquem la col·lecció en un `ArrayList`. Això ens permet mapar l'entrada numèrica de l'usuari (índex) directament a l'objecte seleccionat en temps constant, sense haver d'iterar novament.

2 Visió General del Procés d'Anàlisi

L'objectiu principal del projecte és l'extracció de prototips de comportament a partir de les respostes dels usuaris a una enquesta. Per aconseguir-ho, hem implementat un sistema d'anàlisi que segueix un flux algorítmic de quatre etapes fonamentals.

1. **Codificació (Preprocessing):** El primer repte és convertir les respostes de l'enquesta, que inclouen dades categòriques (selecció única o múltiple), numèriques i de text, en un format numèric homogeni. L'enunciat especifica que hem d'utilitzar el **model d'espai vectorial**. El nostre algorisme de codificació (l'Encoder) s'encarrega de crear una matriu `double[][]` on cada fila representa un usuari i cada columna una característica (feature) derivada de les preguntes.
2. **Selecció de K (Opcional):** Per a algorismes com K-Means, és necessari predefinir el nombre de clústers (k). La nostra implementació inclou el **Mètode del Colze (Elbow Method)** com a funcionalitat opcional per ajudar l'usuari a trobar un valor de k òptim, tal com suggereix la documentació addicional.
3. **Clustering:** Aquest és el nucli de l'anàlisi. Un cop tenim les dades en format vectorial, apliquem algorismes de clustering per agrupar els usuaris. Les funcionalitats principals exigides són els algorismes **K-Means** i **K-Means++**.
4. **Validació:** Finalment, per avaluar d'alguna forma la qualitat del clustering obtingut, hem implementat el **Coeficient de Silhouette**. Aquesta mètrica ens proporciona una puntuació quantitativa de com de ben formats i separats estan els clústers resultants.

Les següents seccions detallen la implementació algorítmica de cadascuna d'aquestes etapes.

3 Algorisme de Codificació (OneHotEncoder)

Fitxers clau: `Encoder/IEncoder.java`, `Encoder/OneHotEncoder.java`

El primer pas, i un dels més crítics, és la transformació de les respostes heterogènies dels usuaris en una representació numèrica unificada: el **model d'espai vectorial**. La nostra implementació, `OneHotEncoder`, s'encarrega d'aquest procés mitjançant una estratègia de *One-Hot Encoding* per a categories i *Bag of Words* per a text.

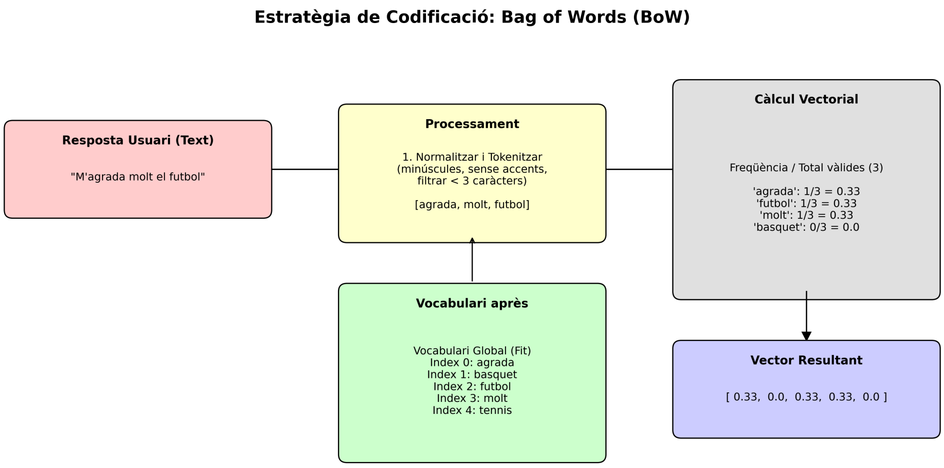

Figura 1: Visualització del procés de codificació de respostes.

El disseny segueix el **Patró Estratègia**, definit per la interfície `IEncoder`.

Justificació de la Implementació: El nostre `OneHotEncoder` es divideix en dues fases: `fit` i `transform`.

- **Fase `fit` (Aprenentatge):** Analitza *totes* les respostes per aprendre l'estructura de les dades.
  - **Categòriques:** Assigna una dimensió única a cada `ChoiceOption` possible.
  - **Numèriques:** Cerca els valors mínims i màxims observats realment en les respostes (no els teòrics). Això és crucial per a una normalització Min-Max efectiva que cobreixi tot el rang [0,1].
  - **Text (Bag of Words):** Per a preguntes obertes de text, construïm un vocabulari de totes les paraules úniques que han utilitzat els usuaris. Primer tokenitzem el text (eliminant puntuació, convertint a minúscules i filtrant paraules molt curtes com connectors) i després assignem una dimensió a cada paraula única del vocabulari global.
- **Fase `transform` (Aplicació):** Converteix cada resposta en un vector.
  - **Categòriques:** Utilitza codificació One-Hot (o Multi-Hot per resposta múltiple), posant un 1.0 a la dimensió corresponent a l'opció triada.
  - **Numèriques:** Aplica la normalització Min-Max: $v' = (v - min)/(max - min)$.
  - **Text:** Utilitzem una representació de freqüència relativa. Per a una resposta donada, comptem quantes vegades apareix cada paraula del vocabulari. Després, normalitzem aquests valors dividint pel nombre total de paraules vàlides en aquella resposta. Això evita que les respostes més llargues tinguin més 'pes' només per la seva longitud, centrant-se en el contingut (freqüència de termes).

Cost Computacional: $O(N \cdot Q \cdot L + V \log V)$ per a `fit` i $O(M \cdot Q \cdot L)$ per a `transform`.

- **Justificació:** El cost està dominat pel processament de text (Bag of Words). En la fase `fit`, l'algorisme recorre les N respostes i Q preguntes, tokenitzant el text (de longitud mitjana L) i ordenant finalment el vocabulari de V paraules ($V \log V$) per garantir determinisme. La transformació de M noves respostes requereix tornar a processar el text linealment ($O(L)$) per comptar freqüències.

El pseudocodi detallat d'aquest procés es troba a l'Annex, secció A.1.

4 Algorisme de Clustering K-Means

Fitxers clau: `kmeans/KMeans.java`, `kmeans/ClusterModel.java`

El nucli del projecte és l'algorisme K-Means. És un algorisme d'aprenentatge no supervisat que agrupa un conjunt de dades X en k clústers. El seu objectiu és minimitzar la **inèrcia**, també coneguda com la Suma d'Errors Quadràtics (SSE), que és la suma de les distàncies al quadrat de cada punt al centroide del seu clúster.

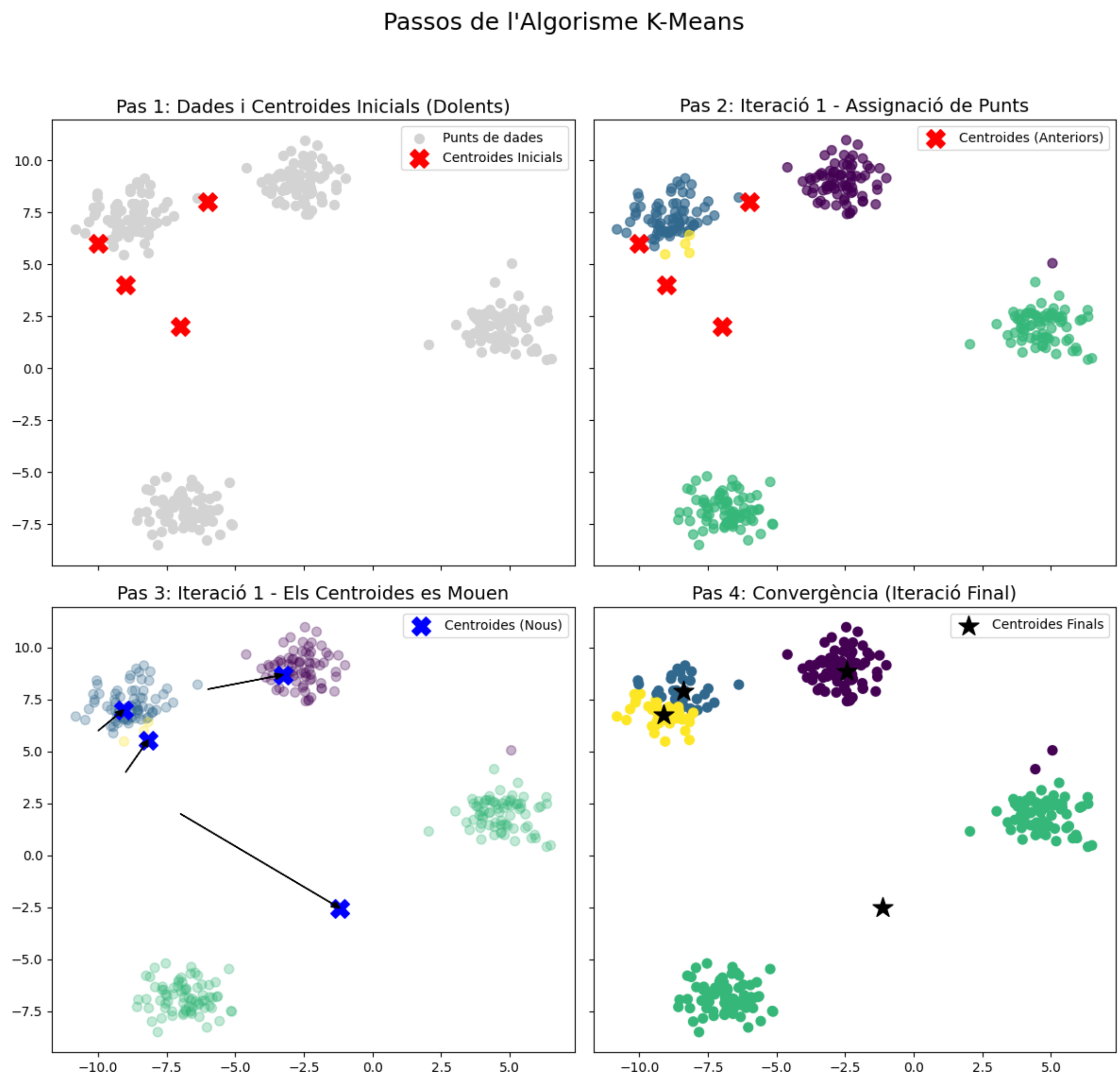

Figura 2: Passos de l'algorisme K-Means (Lloyd).

Justificació de la Implementació: La nostra implementació a `KMeans.java` segueix l'algorisme iteratiu clàssic de Lloyd.

1. **Inicialització (Naive):** (Línies 31-35 a `KMeans.java`). La versió estàndard de K-Means selecciona k punts de dades *aleatòriament* del conjunt X com a centroides inicials. Això es fa barrejant una llista d'índexs i agafant els k primers.

2. **Iteració (E-Step i M-Step):** L'algorisme itera fins que es compleix un criteri d'aturada (nombre màxim d'iteracions `maxIter` o convergència `tol`).
   - **E-Step (Assignació):** (Línies 40-52). Cada punt de dades X_i s'assigna al centroide C_j més proper. Per a això, es calcula la distància (`EuclideanDistance`) de X_i a tots els k centroides i es tria el que té la distància mínima. L'array `labels` emmagatzema l'assignació de clúster per a cada punt.
   - **M-Step (Actualització):** (Línies 59-79). Els centroides es recalculen. El nou centroide de cada clúster es defineix com la *mitjana* (el centre de masses) de tots els punts que hi han estat assignats a l'E-Step.
3. **Gestió de Clústers Buits:** (Línies 66-78). Una part crítica de la nostra implementació és la gestió de clústers que queden buits (cap punt assignat). Si això passa, el centroide "buit"es reassigna a la posició del punt de dades que estigui *més lluny* del seu propi centroide més proper (funció `iFarthest`). Això evita que el nombre de clústers es redueixi i ajuda a moure el centroide a una àrea de possible alta densitat.
4. **Resultat:** El mètode `fit` retorna un objecte `ClusterModel`, que és un contenidor de dades que emmagatzema el resultat final: els centroides, les etiquetes `labels`, la inèrcia final i el nombre d'iteracions.

4.1 Càlcul de Representants

Una part fonamental de l'anàlisi de perfils és determinar quin és el 'prototipus' o representant de cada grup. En la nostra implementació, el representant del clúster és, per definició matemàtica de l'algorisme K-Means, el seu **centroide final**.

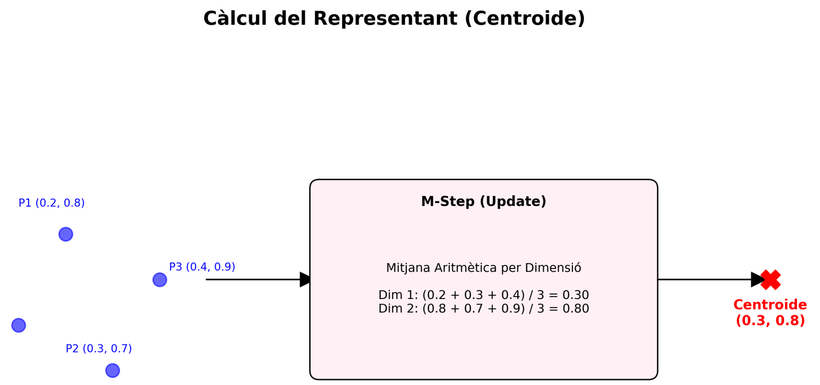

Figura 3: Visualització del càlcul del representant (centroide) com a mitjana.

El càlcul del representant es realitza implícitament durant l'última fase d'actualització (M-Step) de l'algorisme i es troba disponible a l'objecte 'ClusterModel'. Matemàticament, per a un clúster C_j amb $|C_j|$ punts assignats, el vector representant μ_j es calcula com la mitjana aritmètica de tots els vectors x que pertanyen a aquest clúster:

$$\mu_j = \frac{1}{|C_j|} \sum_{x \in C_j} x$$

Això significa que per a cada dimensió del vector (cada característica de l'enquesta), es suma el valor de tots els membres del clúster i es divideix pel nombre total de membres. El vector resultant representa el çomportament promig”d'aquell grup.

Cost Computacional: $O(I \times N \times k \times D)$

- **Justificació:** El cost total és el producte de I (iteracions) pel cost de cada iteració.
- Dins de cada iteració, l'E-Step domina. Per a cadascun dels N punts, hem de calcular la distància a cadascun dels k centroides. Cada càlcul de distància requereix recórrer les D dimensions del vector.
- Per tant, el cost de l'E-Step és $O(N \times k \times D)$.
- L'M-Step (recalcular les mitjanes) només requereix recórrer els N punts un cop per sumar-los ($O(N \times D)$), la qual cosa és menys costosa que l'E-Step.
- El cost total és $I \times (O(N \times k \times D) + O(N \times D)) \approx O(I \times N \times k \times D)$.

El pseudocodi detallat d'aquest procés es troba a l'Annex, secció A.2.

5 Algorisme d'Inicialització K-Means++

Fitxer clau: `kmeans/KMeansPlusPlus.java`

L'algorisme K-Means++ és una de les funcionalitats obligatòries i aborda el principal inconvenient del K-Means estàndard: una mala inicialització aleatòria pot portar a una convergència en un mínim local subòptim (un mal clustering).

K-Means++ no és un algorisme de clustering complet, sinó un mètode **d'inicialització** més intel·ligent. El seu objectiu és triar centroides inicials que estiguin ben distribuïts i allunyats entre si.

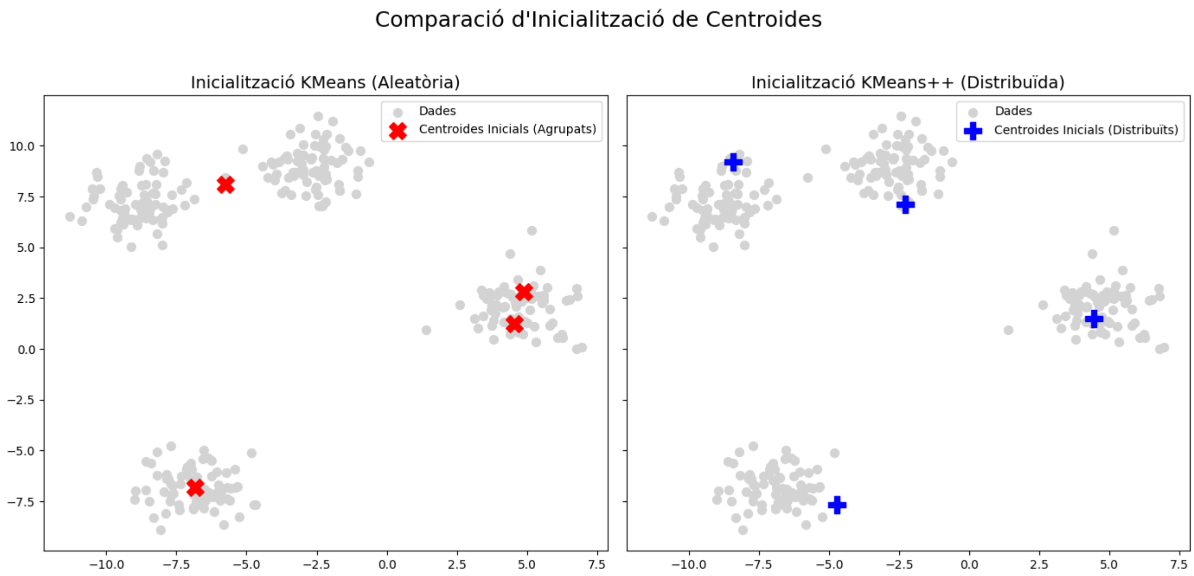

Figura 4: Comparació de la inicialització amb K-Means (esquerra) vs. K-Means++ (dreta).

Justificació de la Implementació:

- **Herència de `KMeans`:** La nostra classe `KMeansPlusPlus` hereta de `KMeans`. Aquesta és una decisió de disseny clau que fomenta la reutilització de codi.
- **Sobreescriptura de `fit`:** `KMeansPlusPlus` sobreescriu el mètode `fit`. Tanmateix, aquest mètode *només* implementa la lògica d'inicialització (línies 24-58 a `KMeansPlusPlus.java`).
- **Lògica d'Inicialització:**
  1. El primer centroide (C_0) es tria de forma aleatòria uniforme, igual que a K-Means.
  2. Per triar els següents $k-1$ centroides, s'utilitza una selecció probabilística ponderada. Per a cada punt X_i, es calcula $D(X_i)^2$, que és la distància al quadrat al centroide *més proper ja seleccionat*.
  3. Els punts amb un $D(X_i)^2$ més alt (és a dir, els que estan més lluny de qualsevol centroide existent) tenen una probabilitat més alta de ser triats com el següent centroide.
- **Reutilització del Codi Base:** Un cop `KMeansPlusPlus.java` ha seleccionat els k centroides inicials, no reimplementa el bucle E-Step/M-Step. En lloc d'això, invoca `super.fitWithCustomInit(...)` (línia 59). Aquest mètode de la classe pare `KMeans` s'encarrega d'executar l'algorisme de Lloyd estàndard, però utilitzant els centroides intel·ligents que li passem com a paràmetre. Això assegura que l'única diferència entre els dos algorismes és la inicialització.

Cost Computacional: $O(N \times D \times k^2 + I \times N \times k \times D)$

- **Justificació:** El cost total és la suma del cost d'inicialització i el cost del K-Means de Lloyd.
- **Cost d'Inicialització ($O(N \times D \times k^2)$):** Per triar cadascun dels k centroides (bucle extern k), hem de recalcular $D(x)^2$ per a tots els N punts. Calcular $D(x)^2$ per a un punt X_i requereix comparar-lo amb tots els c centroides ja triats (on c va fins a k), i cada comparació costa $O(D)$. Això resulta en un cost total per a la inicialització de $\sum_{c=1}^{k-1} O(N \times c \times D) = O(N \times D \times k^2)$.
- **Cost d'Iteració ($O(I \times N \times k \times D)$):** És el cost de l'algorisme K-Means de Lloyd, que s'executa un cop la inicialització ha acabat.

El pseudocodi detallat d'aquest procés d'inicialització es troba a l'Annex, secció A.3.

6 Algorisme de Validació (Silhouette)

Fitxers clau: `validation/IClusterValidation.java`, `validation/Silhouette.java`

Per avaluar d'alguna forma la qualitat del clustering obtingut, la documentació addicional recomana explícitament implementar el **Coeficient de Silhouette**. Aquesta és una mètrica de validació interna (no requereix etiquetes veritables) que mesura com de similars són els punts dins d'un clúster (cohesió) en comparació amb com de diferents són dels punts d'altres clústers (separació).

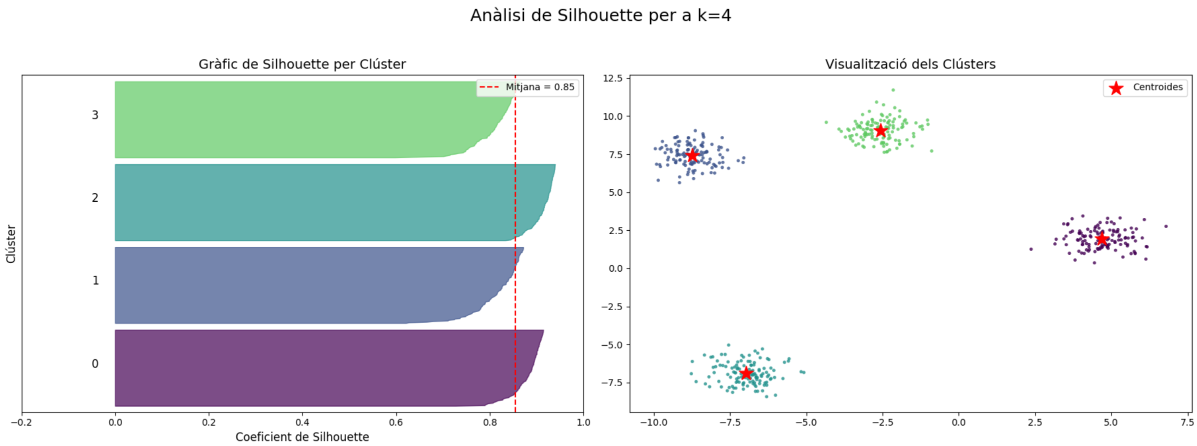

Figura 5: Anàlisi de Silhouette: gràfic per clúster (esquerra) i visualització de dades (dreta).

Justificació de la Implementació: La nostra implementació es troba a `Silhouette.java` i calcula la puntuació per a cada punt i.

- **Càlcul de $a(i)$ (Cohesió):** (Línies 33-41 a `Silhouette.java`). Per a un punt i, $a(i)$ és la *distància mitjana* a tots els altres punts que pertanyen al *mateix clúster*. Un valor baix d'$a(i)$ és desitjable, ja que indica un clúster compacte.

- **Càlcul de $b(i)$ (Separació):** (Línies 43-58). $b(i)$ és la *distància mitjana* al clúster veí més proper. L'algorisme calcula la distància mitjana de i a tots els punts de cada clúster c (on c no és el clúster de i) i $b(i)$ es queda amb el valor mínim d'aquestes mitjanes. Un valor alt de $b(i)$ és desitjable.

- **Puntuació $s(i)$:** (Línia 59). La puntuació final per al punt i s'obté amb la fórmula $s(i) = (b(i) - a(i)) / \max(a(i), b(i))$.
  - $s(i) \approx +1$: Indica que $a(i)$ és molt més petit que $b(i)$. El punt està ben agrupat.
  - $s(i) \approx 0$: Indica $a(i) \approx b(i)$. El punt es troba a la frontera entre dos clústers.
  - $s(i) \approx -1$: Indica $a(i) > b(i)$. El punt està, molt probablement, assignat al clúster incorrecte.

- **Puntuació Mitjana Global:** La interfície `IClusterValidation.java` defineix un mètode `default double average(...)`. Aquest mètode simplement calcula el `scorePerPoint` per a tots els punts i en retorna la mitjana aritmètica. Aquesta mitjana és la que es reporta a l'usuari com a mètrica de qualitat global del clustering.

Cost Computacional: $O(N^2 \times D \times k)$

- **Justificació:** Aquest és un algorisme computacionalment costós. L'operació dominant és el càlcul de $b(i)$.

- Per a cadascun dels N punts (bucle extern):

- 1. Càlcul de $a(i)$: Hem de comparar el punt i amb tots els altres $N - 1$ punts. Cost: $O(N \times D)$.

- 2. Càlcul de $b(i)$: Hem de comparar el punt i amb tots els altres $N - 1$ punts, agrupats pels k clústers. Cost: $O(k \times N \times D)$.

- El cost per a un punt és $O(N \times D \times (1 + k))$.
- El cost total per als N punts és $N \times O(N \times D \times k) = O(N^2 \times D \times k)$. L'alt cost ($O(N^2)$) es deu al fet que la mètrica requereix una comparació de cada punt amb tots els altres punts del conjunt de dades.

El pseudocodi detallat d'aquest procés es troba a l'Annex, secció A.4.

7 Algorisme de Selecció de K (Mètode del Colze)

Fitxers clau: `kselector/IKSelector.java`, `kselector/ElbowMethod.java`

Com a funcionalitat opcional, la documentació addicional suggereix implementar el **Mètode del Colze (Elbow Method)** per a la selecció automàtica de k. Aquest mètode és una heurística que ajuda a trobar un equilibri entre minimitzar la inèrcia i no "sobreajustarl·les dades amb un nombre excessiu de clústers.

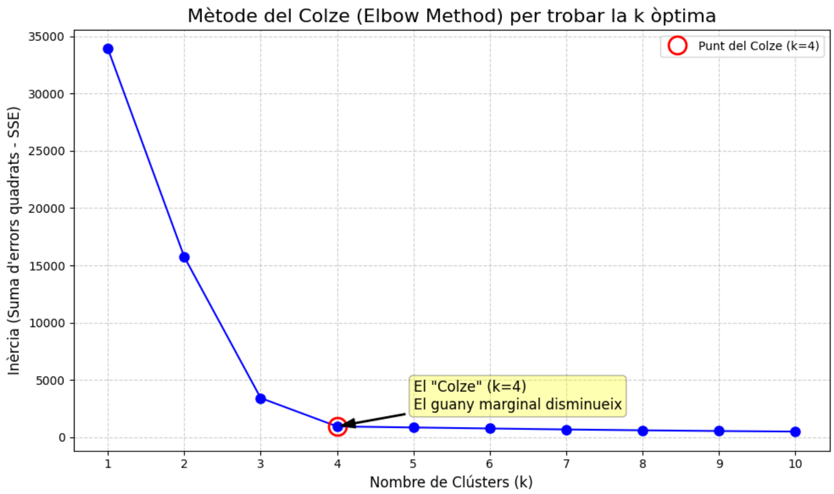

Figura 6: Gràfic del Mètode del Colze, mostrant la inèrcia vs. el nombre de clústers.

Justificació de la Implementació: La nostra implementació a `ElbowMethod.java` (que segueix la interfície `IKSelector`) executa l'algorisme de clustering múltiples vegades.

- **Fase 1: Càlcul d'Inèrcies:** (Línies 23-27 a `ElbowMethod.java`). L'algorisme itera per a cada valor de k en el rang $[kMin, kMax]$. Per a cada k, executa l'algorisme de clustering (p.ex., K-Means) i desa el valor d'inèrcia (SSE) final.
- **Fase 2: Identificació del Colze:** (Línies 29-41). Un error comú és buscar la "major caiguda"d'inèrcia (la qual cosa gairebé sempre afavoreix $k = 2$). La nostra implementació utilitza el mètode geomètric descrit a la documentació:
    1. Es traça una línia recta L des del primer punt $(kMin, \text{inèrcia}(kMin))$ fins a l'últim punt $(kMax, \text{inèrcia}(kMax))$.
    2. Per a cada punt $P_k = (k, \text{inèrcia}(k))$ enmig, es calcula la *distància perpendicular* des de P_k fins a la línia L.
    3. El valor k que maximitza aquesta distància perpendicular és considerat el çolze"(el punt de màxima curvatura) i, per tant, el k òptim suggerit.

  Aquesta implementació és més robusta i troba un equilibri real entre la reducció d'inèrcia i la complexitat del model (nombre de clústers).

Cost Computacional: $O(K_{range} \times (I \times N \times k_{max} \times D))$ o $\approx O(I \times N \times D \times k_{max}^2)$

- **Justificació:** El cost de trobar el punt del colze un cop tenim les inèrcies és negligible ($O(K_{range})$).
- El cost dominant és l'execució de l'algorisme K-Means ($O(I \times N \times k \times D)$) per a cadascun dels K_{range} valors de k (on $K_{range} = k_{max} - k_{min} + 1$).
- El cost total és la suma: $\sum_{k=k_{min}}^{k_{max}} O(I \times N \times k \times D)$.
- Això es pot acotar com K_{range} vegades el cost de l'execució més cara (la de k_{max}), resultant en $O(K_{range} \times I \times N \times k_{max} \times D)$.
- Una expressió més formal de la suma és $O(I \times N \times D \times \sum k) \approx O(I \times N \times D \times k_{max}^2)$.

El pseudocodi detallat d'aquest procés es troba a l'Annex, secció A.5.

A Annex: Pseudocodis dels Algorismes

A.1 Pseudocodi de OneHotEncoder.transform (actualitzat)

Aquest pseudocodi il·lustra com es construeix un únic vector per a una resposta d'usuari, incloent la lògica de "Bag of Words"per a text.

Algorithm 1 OneHotEncoder.transform

```
function TransformRow(resposta, diccionaris)
    vector ← nou array de double[totalDims]                    ▷ Inicialitzat a 0.0
    for all pregunta q in enquesta.getQuestionsOrdenades() do
        ans ← resposta.getAnswer(q.getId())
        if ans is null or empty then continue
        end if
        if q is Categorical (Single/Multi) then
            for all idOpcio in ans.getSelectedIds() do
                idx ← diccionaris.categorial.get(q.getId(), idOpcio)
                vector[idx] ← 1.0
            end for
        else if q is OpenIntQuestion then
            idx ← diccionaris.numeric.get(q.getId())
            [min, max] ← diccionaris.dominis.get(q.getId())
            vector[idx] ← (ans.getValue() − min)/(max − min)
        else if q is OpenStringQuestion (Text) then
            startIdx ← diccionaris.textStart.get(q.getId())
            mapaParaules ← diccionaris.textVocab.get(q.getId())
            tokens ← tokenize(ans.getValue())          ▷ minúscules, filter més de 2 chars
            freqLocal ← nou Mapa
            totalValid ← 0
            for all w in tokens do
                if mapaParaules.containsKey(w) then
                    absIdx ← startIdx + mapaParaules.get(w)
                    freqLocal.put(absIdx, freqLocal.getOrDefault(absIdx, 0) + 1)
                    totalValid ← totalValid + 1
                end if
            end for
                                                ▷ Normalització per freqüència relativa
            if totalValid > 0 then
                for all (idx, count) in freqLocal do
                    vector[idx] ← count/totalValid
                end for
            end if
        end if
    end for
    return vector
end function
```

A.2 Pseudocodi de l'Algorisme K-Means (Lloyd)

Aquest pseudocodi mostra el procés iteratiu de l'algorisme K-Means, incloent l'assignació (E-Step), l'actualització (M-Step) i la gestió de clústers buits.

Algorithm 2 Algorisme K-Means (Lloyd)

```
function FIT(Dades X, Enter k, Distància dist, Seed seed, MaxIter maxIter, Tol tol)
    n ← X.length, d ← X[0].length
    C ← inicialitzacioAleatoria(X, k, seed)                    ▷ Tria k punts únics de X
    labels ← nou array d'enters[n]
    inerciaPrevia ← ∞
    for it ← 1 to maxIter do
        inerciaActual ← 0.0
                                                              ▷ — E-Step (Assignació) —
        for i ← 0 to n − 1 do
            distMin ← ∞, clusterProper ← −1
            for j ← 0 to k − 1 do
                distancia ← dist.between(X[i], C[j])
                if distancia < distMin then
                    distMin ← distancia, clusterProper ← j
                end if
            end for
            labels[i] ← clusterProper
            inerciaActual ← inerciaActual + (distMin^2)
        end for
                                                  ▷ — Comprovació de Convergència —
        if |inerciaPrevia − inerciaActual| ≤ tol then
            break                                              ▷ Convergència assolida
        end if
        inerciaPrevia ← inerciaActual
                        ▷ — M-Step (Actualització / Càlcul de Representants) —
        C_nou ← nou array double[k][d]
        comptadors ← nou array d'enters[k]
        for i ← 0 to n − 1 do
            c ← labels[i]
            C_nou[c] ← C_nou[c] + X[i]                                ▷ Suma vectorial
            comptadors[c] ← comptadors[c] + 1
        end for
        for j ← 0 to k − 1 do
            if comptadors[j] > 0 then
                C_nou[j] ← C_nou[j]/comptadors[j]              ▷ Mitjana = Representant
            else
                C_nou[j] ← X[iFarthest(X, C, dist)]               ▷ Gestió clúster buit
            end if
        end for
        C ← C_nou
    end for
    return new ClusterModel(C, labels, inerciaPrevia, it)
end function
```

A.3 Pseudocodi de la Inicialització K-Means++

Aquest pseudocodi descriu el mètode de selecció de centroides inicials de K-Means++. L'objectiu és triar centroides que estiguin allunyats entre si mitjançant una selecció probabilística ponderada.

Algorithm 3 Inicialització K-Means++

```
function InitializeKPP(Dades X, Enter k, Seed seed)
    C ← nou array double[k][d]
    C[0] ← X[rnd.nextInt(n)]                              ▷ 1. El primer centroide és aleatori
    distanciesQuadrades ← nou array double[n]
    for c ← 1 to k − 1 do
        sumaDistQuad ← 0.0                     ▷ 2. Calcular D(x) al cuadrat per a cada punt
        for i ← 0 to n − 1 do
            distMinQuad ← distància al quadrat a C[0]
            for j ← 1 to c − 1 do                  ▷ Dist. al centroide més proper JA TRIAT
                distMinQuad ← min(distMinQuad, distància al quadrat a C[j])
            end for
            distanciesQuadrades[i] ← distMinQuad
            sumaDistQuad ← sumaDistQuad + distMinQuad
        end for                                                  ▷ 3. Selecció ponderada
        valorAleatori ← rnd.nextDouble() ∗ sumaDistQuad
        sumaAcumulada ← 0.0, puntTriat ← −1
        for i ← 0 to n − 1 do
            sumaAcumulada ← sumaAcumulada + distanciesQuadrades[i]
            if sumaAcumulada ≥ valorAleatori then
                puntTriat ← i
                break
            end if
        end for
        C[c] ← X[puntTriat]
    end for
    return C                                          ▷ Centroides inicials per a K-Means
end function
```

A.4 Pseudocodi del Coeficient de Silhouette (per punt)

Aquest pseudocodi detalla el càlcul de la puntuació de Silhouette per a un únic punt i, calculant la seva cohesió interna ($a(i)$) i la seva separació respecte al clúster veí més proper ($b(i)$).

Algorithm 4 Coeficient de Silhouette (per punt)

```
function SCOREPERPOINT(Dades X, Labels labels, Distància dist)
    n ← X.length, k ← max(labels) + 1
    s ← nou array double[n]
    for i ← 0 to n − 1 do
        clusterActual ← labels[i]
                                                ▷ — Càlcul de a(i) [Cohesió] —
        a ← 0.0, comptadorA ← 0
        for j ← 0 to n − 1 do
            if i ≠ j and labels[j] = clusterActual then
                a ← a + dist.between(X[i], X[j])
                comptadorA ← comptadorA + 1
            end if
        end for
        a ← if comptadorA > 0 then a/comptadorA else 0
                                                ▷ — Càlcul de b(i) [Separació] —
        b ← ∞
        for c ← 0 to k − 1 do
            if c = clusterActual then continue
            end if
            sumaDistVeina ← 0.0, comptadorB ← 0
            for j ← 0 to n − 1 do
                if labels[j] = c then
                    sumaDistVeina ← sumaDistVeina + dist.between(X[i], X[j])
                    comptadorB ← comptadorB + 1
                end if
            end for
            if comptadorB > 0 then
                distMitjanaVeina ← sumaDistVeina/comptadorB
                if distMitjanaVeina < b then b ← distMitjanaVeina
                end if
            end if
        end for
        if b = ∞ then b ← 0
        end if                                  ▷ Cas k=1
                                                ▷ — Càlcul de s(i) —
        s[i] ← (b − a)/ max(a, b)
        if max(a, b) = 0 then s[i] ← 0
        end if
    end for
    return s
end function
```

A.5 Pseudocodi del Mètode del Colze

Aquest pseudocodi implementa el mètode geomètric per trobar el çolze". Primer, executa K-Means per a un rang de k i desa les inèrcies. Després, troba el punt k que té la distància perpendicular més gran a la línia que uneix la primera i l'última inèrcia.

Algorithm 5 Mètode del Colze

```
function SuggestK(Dades X, kMin, kMax, Algorisme algo, ...)
    inercies ← nova Llista
    valorsK ← nova Llista
                                        ▷ 1. Executar K-Means per a cada k
    for k ← kMin to kMax do
        model ← algo.fit(X, k, ...)
        inercies.add(model.getInertia())
        valorsK.add(k)
    end for
                                        ▷ 2. Trobar el punt més allunyat de la línia
    P1 ← (kMin, inercies[0])
    P2 ← (kMax, inercies.last())
    distMax ← −1, kOptima ← kMin
    for i ← 0 to inercies.size() − 1 do
        P0 ← (valorsK[i], inercies[i])
        distancia ← distanciaPerpendicular(P0, linia(P1, P2))
        if distancia > distMax then
            distMax ← distancia
            kOptima ← valorsK[i]
        end if
    end for
    return kOptima
end function
```