Documentació d'Estructures de Dades i Algorismes

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

Lliurament 1.0 - Tardor 2025-26

Índex

1 Estructures de Dades Clau del Projecte

Per implementar la lògica del projecte, s'han seleccionat estructures de dades de les llibreries estàndard de Java. Ens hem centrat 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<Integer, Integer>> categoricalVocab`, `Map<Integer, Integer> numericFeatureMap`, `Map<Integer, double[]> numericDomains`
    - **Implementació:** `HashMap`
    - **Cost:** `put` (en `fit`) i `get` (en `transform`) són $O(1)$ en mitjana.
    - **Justificació:** Durant la transformació (mètode `transform`), per a cada resposta d'usuari, hem de buscar ràpidament la informació de codificació (l'índex de columna, el rang [min, max], etc.) associada a l'ID de la pregunta. El `HashMap` proporciona aquesta cerca en $O(1)$ (temps mitjà) basant-se en el hash de l'ID de la pregunta.

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.

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**. Els algorismes de clustering com K-Means no poden operar directament sobre text o categories; requereixen vectors de nombres. La nostra implementació, `OneHotEncoder`, s'encarrega d'aquest procés.

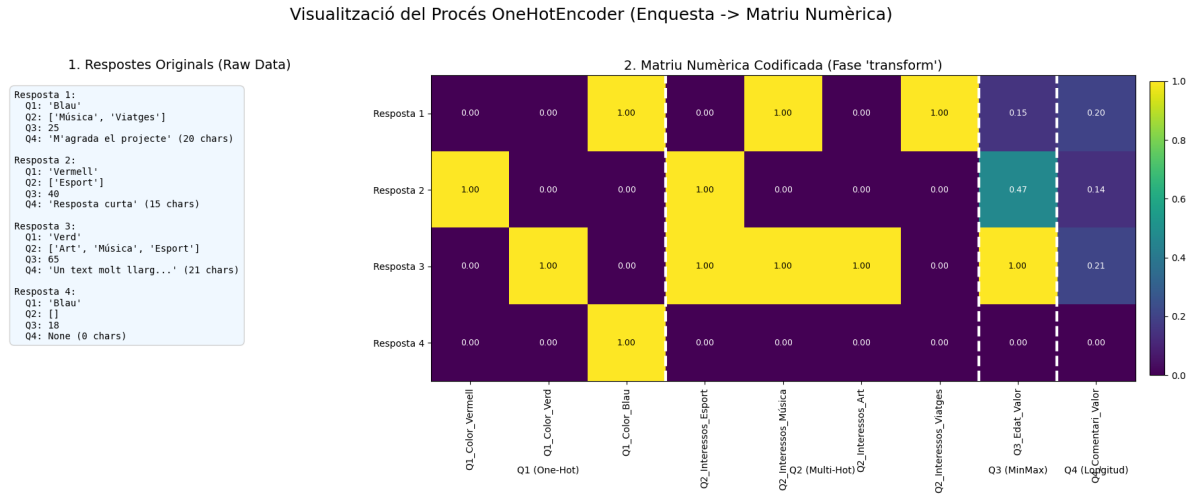

Figura 1: Visualització del procés de codificació de respostes.

El disseny segueix el **Patró Estratègia**, definit per la interfície `IEncoder`. Això permetria canviar el mètode de codificació sense modificar els controladors d'anàlisi.

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

- **Fase `fit` (Aprenentatge):** Aquest mètode (línies 48-112 a `OneHotEncoder.java`) aprèn l'estructura de la matriu de dades. Itera sobre totes les preguntes del `Survey` i construeix un "diccionari"de característiques.
  - **Categòriques (`SingleChoice`, `MultipleChoice`):** S'adopta la codificació *One-Hot* (i *Multi-Hot* per a múltiple). Cada `ChoiceOption` possible (ex: "Vermell", "Verd") es converteix en una columna (dimensió) única a la matriu. Això és fonamental per tractar dades nominals sense introduir un ordre artificial.
  - **Numèriques (`OpenInt`):** Per a normalitzar correctament, no n'hi ha prou amb el rang teòric de la pregunta (ex: 0-100). L'algorisme `fit` itera sobre *totes* les respostes dels usuaris per trobar el valor mínim i màxim *observat* realment. Aquests valors `[min, max]` es guarden al mapa `numericDomains` i són crucials per a una normalització Min-Max efectiva durant la transformació.
  - **Text (`OpenString`):** Com a simplificació inicial (amb possibles millores a lliuraments futurs), hem optat per convertir el text en un valor numèric que representa la seva **longitud** (línies 101-103). Aquest valor es normalitza després dividint-lo pel `maxLength` de la pregunta. És una heurística que captura una característica simple del text (esforç o detall) de manera computacionalment eficient.
- **Fase `transform` (Aplicació):** Aquest mètode (línies 126-191) aplica el diccionari après per construir la matriu `double[][]`. Per a cada `SurveyResponse`, crea un vector.
  - Per a `SingleChoice`, posa un `1.0` a la columna de l'opció triada i `0.0` a les altres.
  - Per a `MultipleChoice`, posa `1.0` a totes les columnes de les opcions triades.
  - Per a `OpenInt`, aplica la normalització Min-Max: $valorNormalitzat = (valor - min_{obs})/(max_{obs} - min_{obs})$. Això escala el valor a l'interval [0, 1].
  - **Gestió de dades absents:** Si una pregunta no va ser contestada (`ans == null` o `ans.isEmpty()`), totes les columnes associades a aquesta pregunta es queden amb el seu valor per defecte, `0.0`. Aquesta és una forma simple i efectiva d'imputació (assignar un valor a les dades que falten).

Cost Computacional: Sigui N el nombre de respostes (`SurveyResponse`) i P el nombre de preguntes (`Question`).

- **Cost de `fit`:** $O(P \times O_{avg} + P_{num} \times N \times A_{avg})$
- **Justificació:** El cost es compon de dues parts. Primer, recórrer les P preguntes per construir el vocabulari de les categòriques, que depèn del nombre total d'opcions ($P \times O_{avg}$). Segon, i més costós, per a cada pregunta numèrica (P_{num}), ha de recórrer les N respostes, i dins de cadascuna, les A_{avg} respostes individuals per trobar el rang [min, max].
- **Cost de `transform`:** $O(N' \times (P + P_{multi} \times M_{avg}))$

- **Justificació:** Per a cadascuna de les N' respostes a transformar, hem de recórrer les P preguntes. Per a la majoria de preguntes, la cerca al `HashMap` i l'assignació és $O(1)$. Tanmateix, per a les preguntes de resposta múltiple (P_{multi}), hem d'iterar addicionalment per les M_{avg} opcions seleccionades per l'usuari.

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.

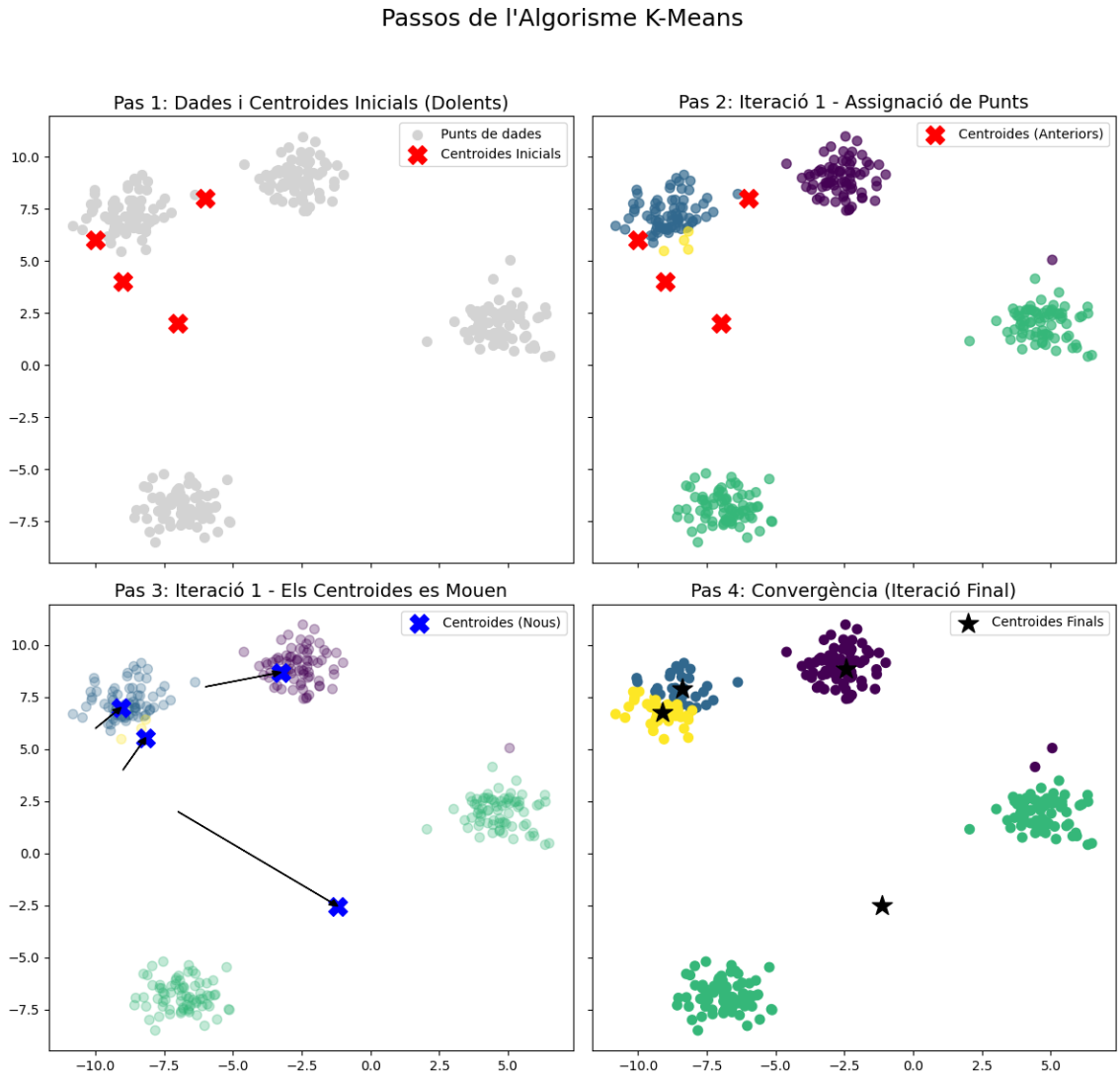

Figura 2: Passos de l'algorisme K-Means (Lloyd).

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.

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.

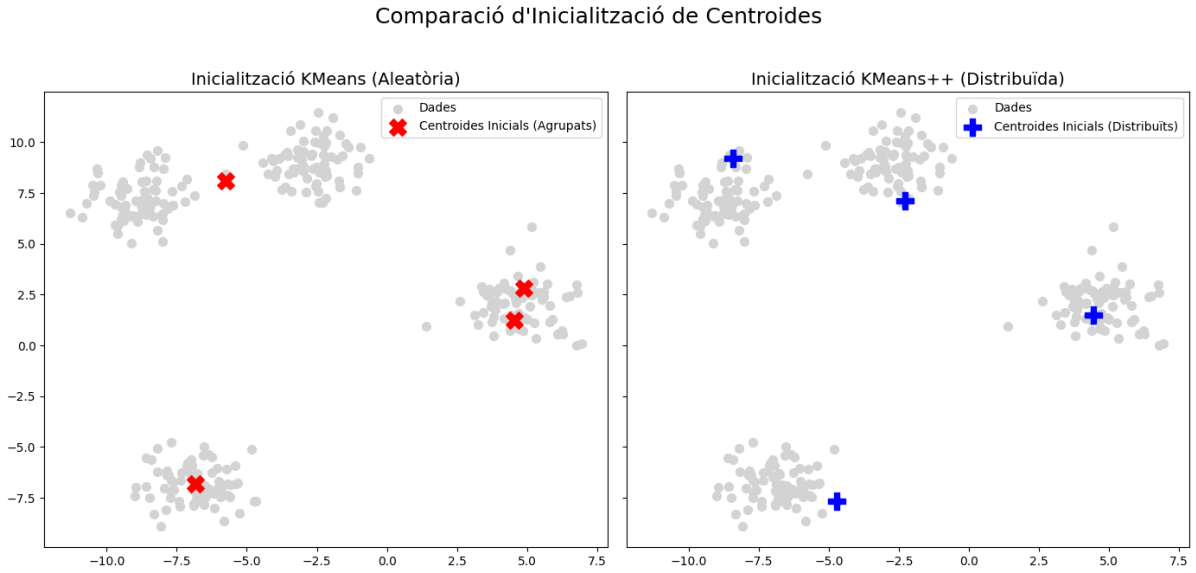

Figura 3: Comparació de la inicialització amb K-Means (esquerra) vs. K-Means++ (dreta).

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).

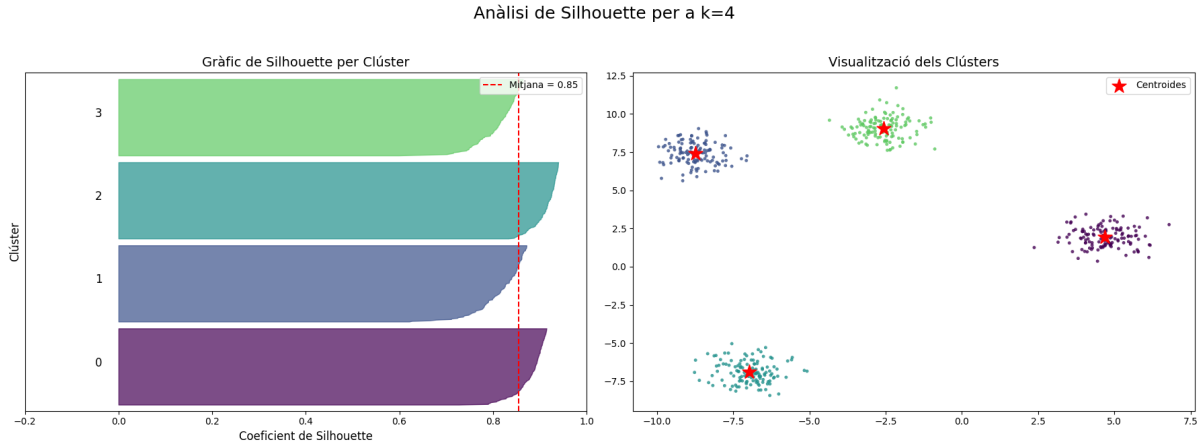

Figura 4: Anàlisi de Silhouette: gràfic per clúster (esquerra) i visualització de dades (dreta).

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.

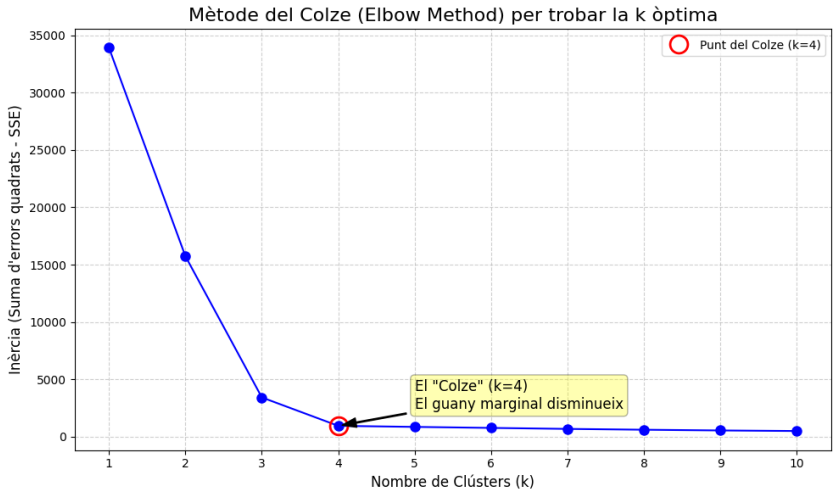

Figura 5: Gràfic del Mètode del Colze, mostrant la inèrcia vs. el nombre de clústers.

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 tracça 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

Aquest pseudocodi il·lustra com es construeix un únic vector (fila) per a una resposta d'usuari, basant-se en els diccionaris creats durant la fase `fit`.

Algorithm 1 OneHotEncoder.transform (per a una resposta)

```
function TransformRow(resposta, diccionaris)
    vector ← nou array de double[totalDims]                    ▷ Inicialitzat a 0.0
    mapaRespostes ← mapeja(resposta.getAnswers()) per ID de pregunta
    for all pregunta q in enquesta.getQuestionsOrdenades() do
        respostaUsuari ← mapaRespostes.get(q.getId())
        if respostaUsuari is null or respostaUsuari.isEmpty() then
            continue                                           ▷ Deixa els valors com a 0.0
        end if
        if q is SingleChoiceQuestion then
            idOpcio ← respostaUsuari.getOptionId()
            indexCol ← diccionaris.categorial.get(q.getId(), idOpcio)
            vector[indexCol] ← 1.0
        else if q is MultipleChoiceQuestion then
            for all idOpcio in respostaUsuari.getOptionIds() do
                indexCol ← diccionaris.categorial.get(q.getId(), idOpcio)
                vector[indexCol] ← 1.0
            end for
        else if q is OpenIntQuestion then
            indexCol ← diccionaris.numeric.get(q.getId())
            valor ← respostaUsuari.getValue()
            [min, max] ← diccionaris.dominis.get(q.getId())
            vector[indexCol] ← (valor − min)/(max − min)       ▷ Normalització
        else if q is OpenStringQuestion then
            indexCol ← diccionaris.numeric.get(q.getId())
            longitud ← respostaUsuari.getValue().length()
            [min, maxLen] ← diccionaris.dominis.get(q.getId())
            vector[indexCol] ← longitud/maxLen                 ▷ Normalització
        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 + (distMin²)
        end for
                                                           ▷ — Comprovació de Convergència —
        if |inerciaPrevia − inerciaActual| ≤ tol then
            break                                          ▷ Convergència assolida
        end if
        inerciaPrevia ← inerciaActual
                                                           ▷ — M-Step (Actualització) —
        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]          ▷ Càlcul de la mitjana
            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
```