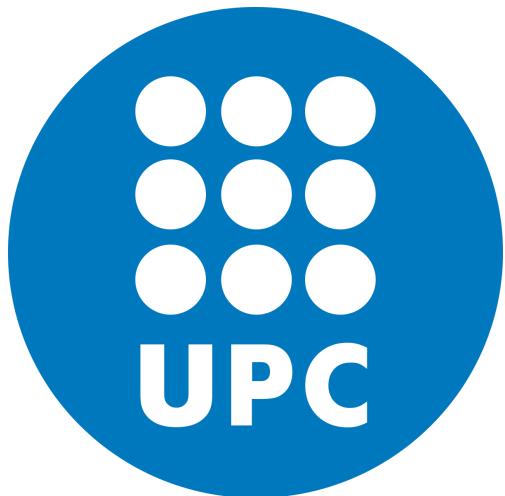


Extractor de prototipos de comportamiento o perfiles

Documentación

Versión 2.0

Proyectos de Programación QT 2025-26



Grupo 22-4

Hadeer Abbas Khalil Wysocka - hadeer.abbas.khalil

Yimin Jin - yimin.jin

Sergi Malaguilla Bombín - sergi.malaguilla

Javier Zhangpan - javier.zhangpan

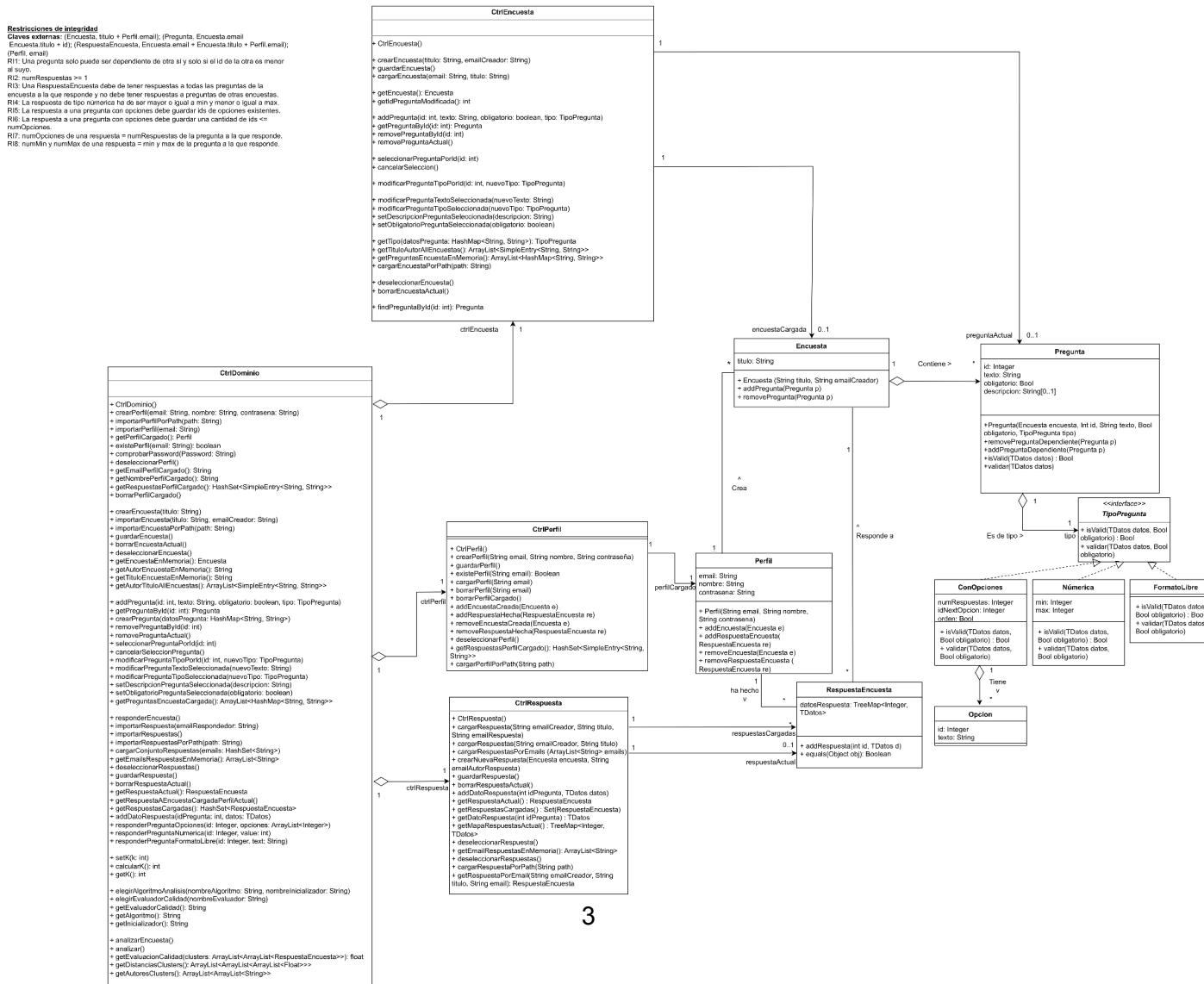
Andrés Lafuente Patau - andres.lafuente

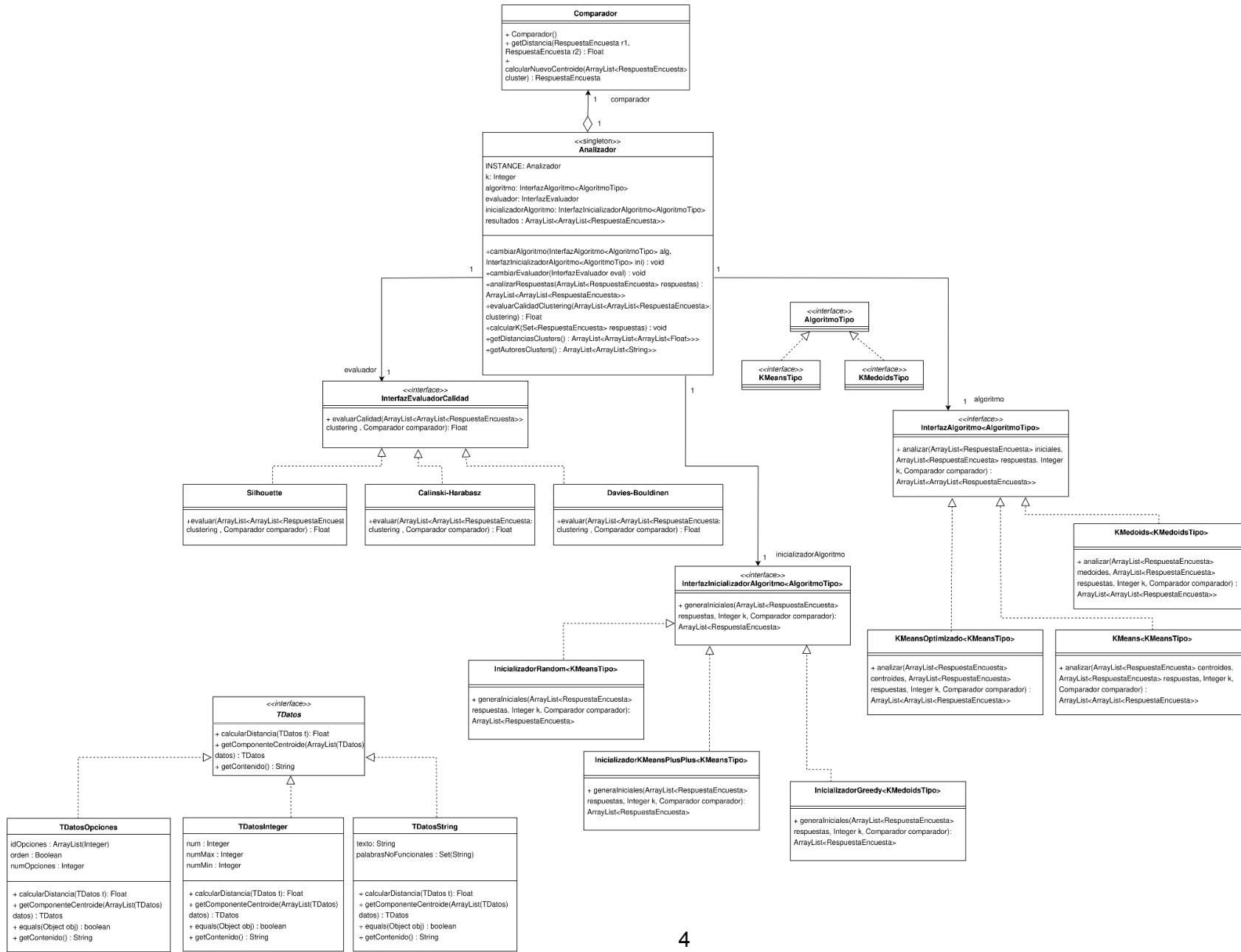
Índice

1. Diagrama de la capa de Dominio	3
1.1 Descripción de las clases de Dominio	5
2. Diagrama de la capa de Persistencia	13
2.1 Estructura de los ficheros	14
2.2 Descripción de las clases de Persistencia	15
3. Diagrama de la capa de Presentación	18
3.1 Descripción de las clases de Presentación	19
4. Estructuras de datos y algoritmos usados	24
4.1. Estructuras de Datos	24
4.1.1 HashSet	24
4.1.2 ArrayList	24
4.1.3 TreeMap	25
4.1.4 SimpleEntry	26
4.1.5 HashMap	27
4.2 Algoritmos	27
4.2.1 K-Means	27
4.2.3 K-Means-Optimizado	30
4.2.2 Inicialización Random	31
4.2.3 K-Means++	32
4.2.4 K-Medoids	33
4.2.5 Inicialización Greedy	35
4.2.6 Silhouette	36
4.2.7 Calinski-Harabasz	37
4.2.8 Davies-Bouldin	37

1. Diagrama de la capa de Dominio

Adjuntamos el diagrama en la carpeta DOCS. Hemos obviado los getters y setters.





1.1 Descripción de las clases de Dominio

Perfil

La clase Perfil representa a los usuarios de la aplicación de encuestas. Cada perfil tiene un nombre de usuario, contraseña y correo. El correo identifica cada perfil. El perfil también puede tener guardado sus encuestas creadas y sus respuestas a encuestas.

Encuesta

La clase Encuesta representa las encuestas. Cada encuesta es identificada por su título y su creador. Las encuestas guardan sus preguntas y las gestionan.

Pregunta

La clase Pregunta representa una pregunta de una encuesta. Cada pregunta almacena su identificador, texto y otros atributos como la obligatoriedad. Cada pregunta tiene asociada un TipoPregunta, que contiene la información sobre el tipo de la pregunta.

RespuestaEncuesta

La clase RespuestaEncuesta representa una respuesta a una encuesta. Esta clase se identifica por el respondedor y la encuesta a la que responde. Cada RespuestaEncuesta guarda un TreeMap de TDatos, que son las respuestas a preguntas individuales de la encuesta.

Analizador

La clase Analizador es quien se encarga de toda la gestión de la algoritmia implicada en el clustering de las respuestas. Contiene: un comparador cuyo uso se emplea más adelante en los propios algoritmos, un evaluador de calidad (de diferentes tipos) que mediante un algoritmo determina un valor que indica su calidad, y un algoritmo de clustering (de dos tipos) que realiza la funcionalidad principal de analizar las respuestas y devolver los clusters resultantes, junto a su inicializador compatible correspondiente. Se le ha aplicado el patrón Singleton al

Analizador para poder instanciar el mismo objeto desde donde sea necesario y no tener que guardar diferentes instancias, ya que solo hace falta uno para realizar el análisis.

TipoPregunta

TipoPregunta es una interfaz que contiene los métodos isValid y validar. Esta interfaz la hemos creado aplicando el patrón estrategia para conseguir dos objetivos: poder modificar el tipo de la pregunta en tiempo de ejecución y tener diferentes estrategias de validación para cada tipo separado de la clase Pregunta. De esta manera, tanto la clase Pregunta como otras clases que necesiten validar TDatos no necesitan saber la implementación concreta de TipoPregunta, mejorando el acoplamiento y la escalabilidad.

ConOpciones

La clase ConOpciones es una implementación de TipoPregunta. Esta clase representa un tipo de múltiples opciones, guardando un ArrayList de Opcion. Esta clase se encarga de validar TDatosOpciones, implementando los métodos isValid y validar de TipoPregunta.

Numerica

La clase Numerica es una implementación de TipoPregunta. Esta clase representa un tipo de pregunta que acepta números enteros, guardando el rango de estos números. Esta clase se encarga de validar TDatosInteger, implementando los métodos isValid y validar de TipoPregunta.

FormatoLibre

La clase FormatoLibre es una implementación de TipoPregunta. Esta clase representa un tipo de pregunta con respuesta libre. Esta clase se encarga de validar TDatosString, implementando los métodos isValid y validar de TipoPregunta.

Opcion

La clase Opcion representa una opción en una pregunta de múltiples opciones. Esta clase almacena el identificador de la opción y su texto.

Comparador

La clase Comparador contiene el método para calcular la distancia entre dos RespuestaEncuesta y el método para calcular un nuevo centroide en un ArrayList de RespuestaEncuesta. Esta clase la hemos creado para no tener estas funcionalidades en la clase RespuestaEncuesta.

InterfazEvaluadorCalidad

InterfazEvaluadorCalidad es una interfaz que tiene el método para evaluar la calidad de un clustering. Esta interfaz la hemos creado aplicando el patrón estrategia, permitiendo modificar el algoritmo que el analizador debe utilizar en tiempo de ejecución y facilitar la creación de nuevos algoritmos.

Silhouette

La clase Silhouette implementa InterfazEvaluadorCalidad. Esta clase utiliza el Coeficiente de Silhouette para evaluar un clustering.

Calinski-Harabasz

La clase Calinski-Harabasz implementa InterfazEvaluadorCalidad. Esta clase utiliza el Índice de Calinski-Harabasz para evaluar un clustering.

Davies-Bouldinen

La clase Davies-Bouldinen implementa InterfazEvaluadorCalidad. Esta clase utiliza el Índice de Davies-Bouldin para evaluar un clustering.

TDatos

TDatos es una interfaz que hemos creado para abstraer el formato de las respuestas a preguntas, aplicando el patrón estrategia. Esta interfaz también tiene los métodos para calcular distancias entre dos TDatos y calcular el centroide de un

ArrayList de TDatos. De esta manera, otras clases no necesitan conocer los TDatos concretos.

TDatosOpciones

La clase TDatosOpciones representa las opciones elegidas en una respuesta a una pregunta de tipo Opcion. La clase contiene atributos como “idOpciones” que contiene las opciones propiamente escogidas, “orden” para saber si las opciones tienen un orden asociado entre ellas y “numOpciones” que indica el número total de opciones que había en la pregunta. La clase también ofrece métodos para comparar, calcular distancias y calcular centroides de datos de este tipo.

TDatosInteger

La clase TDatosInteger representa el número entero escrito en una respuesta a una pregunta de tipo Numerica. La clase contiene atributos como “num”, que es la propia respuesta y “numMax” y “numMin” que son los valores máximos permitidos por la pregunta. La clase también ofrece métodos para comparar, calcular distancias y calcular centroides de datos de este tipo.

TDatosString

La clase TDatosString representa un texto escrito en una respuesta a una pregunta de tipo FormatoLibre. La clase contiene atributos como “texto”, que es la propia respuesta y “palabrasNoFuncionales” que es un listado de palabras sin significado semántico propio de la lengua española. La clase también ofrece métodos para comparar, calcular distancias y calcular centroides de datos de este tipo.

AlgoritmoTipo

AlgoritmoTipo es una interfaz de marcador cuyo propósito es diferenciar los inicializadores exclusivos a ciertos algoritmos. Esta interfaz permite imposibilitar la creación de combinaciones que consideramos no aptas para su uso y mejora la legibilidad del código fuente.

InterfazInicializador<AlgoritmoTipo>

InterfazInicializador es una interfaz que tiene el método para inicializar los centroides. Esta interfaz la hemos creado aplicando el patrón estrategia, permitiendo modificar el algoritmo que el analizador debe utilizar en tiempo de ejecución y facilitar la creación de nuevos algoritmos de inicialización.

InicializadorKMeansPlusPlus<KMeansTipo>

La clase InicializadorKMeansPlusPlus implementa un inicializador del algoritmo K-Means que sigue el algoritmo de K-Means++. La clase solamente contiene un atributo de la clase Random presente en las librerías de Java para generar números aleatorios. La clase ofrece un método para generar centroides iniciales dadas unas respuestas.

InicializadorRandom<KMeansTipo>

La clase InicializadorRandom implementa un inicializador del algoritmo K-Means que sigue un algoritmo trivial aleatorio. La clase solamente contiene un atributo de la clase Random presente en las librerías de Java para generar números aleatorios. La clase ofrece un método para generar centroides iniciales dadas unas respuestas.

InicializadorGreedy<KMedoidsTipo>

La clase InicializadorGreedy implementa un inicializador para el algoritmo k-medoids, basado en la estrategia Greedy. Su función es seleccionar un número de medoids de forma determinista a través de la minimización de las distancias entre respuestas y los candidatos a medoid para elegir los medoids iniciales.

InterfazAlgoritmo<AlgoritmoTipo>

InterfazAlgoritmo es una interfaz que tiene el método para analizar un ArrayList de RespuestaEncuesta. Esta interfaz la hemos creado aplicando el patrón estrategia, permitiendo modificar el algoritmo que el analizador debe utilizar en tiempo de ejecución y facilitar la creación de nuevos algoritmos.

KMeans<KMeansTipo>

La clase KMeans implementa el algoritmo de clustering “naïve” de K-Means, también llamado algoritmo de Lloyd. La clase no contiene ningún atributo y simplemente ofrece un método que clasifica las respuestas dadas en un número de clusters diferentes. Se ha de tener en cuenta que es posible que se devuelvan clusters vacíos en casos extremos, pero se asegura que todas respuestas formarán parte de algún cluster.

KMeansOptimizado<KMeansTipo>

La clase KMeansOptimizado implementa el algoritmo de clustering KMeans explotando la propiedad de desigualdad triangular, ahorrando cómputo de distancias, aunque a cambio requiere mayor capacidad en memoria. La clase no contiene ningún atributo y ofrece el mismo método que el KMeans anterior. De manera análoga, también puede producir particiones vacías en algunos casos.

KMedoids<KMedoidsTipo>

La clase KMedoids implementa el algoritmo alternativo de clustering de las respuestas de una misma encuesta, recibe como parámetro unos medoids predeterminados/inicializados por el algoritmo greedy que son un subconjunto de las respuestas a las cuales queremos aplicar el clustering, que también pasamos como parámetro; junto a un entero que indica el número de clusters que ha de devolver. El algoritmo devuelve un ArrayList de clusters que en si son ArrayLists de Respuestas.

CtrlEncuesta

La clase CtrlEncuesta implementa la gestión de las diferentes encuestas y mantiene una encuesta cargada posibilitando así la modificación de una encuesta cargada por el usuario. Se encarga de coordinar las operaciones sobre una encuesta, asegurando que todas las acciones sean válidas y mantengan la integridad de los datos. Las diferentes responsabilidades del controlador es la creación, carga, guardado, selección/deselección y modificación de una encuesta, esta última ofrece la posibilidad de añadir, eliminar y modificar preguntas individuales que el usuario puede ir seleccionando para acceder a cualquiera de ellas a través de su ID. También ofrece getters para que los diferentes drivers puedan consultar la diferente información de una encuesta y sus preguntas.

CtrlPerfil

La clase CtrlPerfil se encarga de hacer toda la gestión relacionada con los casos de uso donde se involucran los perfiles, contiene un único perfil que indica el perfil que está cargado en el sistema, permitiendo así que el usuario pueda responder y crear encuestas con su perfil asociado. Aporta funcionalidades para crear, cargar, guardar perfiles y gestionar las asociaciones que se generan después de crear y/o responder encuestas (precisamente la que está cargada en el CtrlEncuesta).

CtrlRespuesta

La clase CtrlRespuesta es la responsable de la gestión de todas las operaciones relacionadas con las respuestas de la clase RespuestaEncuesta. Contiene una respuesta que indica la respuesta actual que se está modificando y un conjunto de respuestas para que se pueda realizar el análisis. Aporta métodos para crear, cargar, guardar y borrar respuestas, y añadir los datos adecuados correspondientes a la pregunta de la encuesta cargada en el CtrlEncuesta en la respuesta actual cargada.

CtrlDominio

Se trata del controlador principal de la capa de dominio y su función es coordinar las operaciones relacionadas con la gestión de encuestas, perfiles, respuestas y análisis. Actúa como intermediario entre la “interfaz” y el dominio. Ofrece un punto de entrada unificado para implementar los casos de uso delegando las diferentes implementaciones a los controladores específicos CtrlEncuesta, CtrlPerfil, CtrlRespuesta y el singleton Analizador, sin implementar lógica compleja a sí mismo. Éste controlador desarrolla las funcionalidades principales que serían:

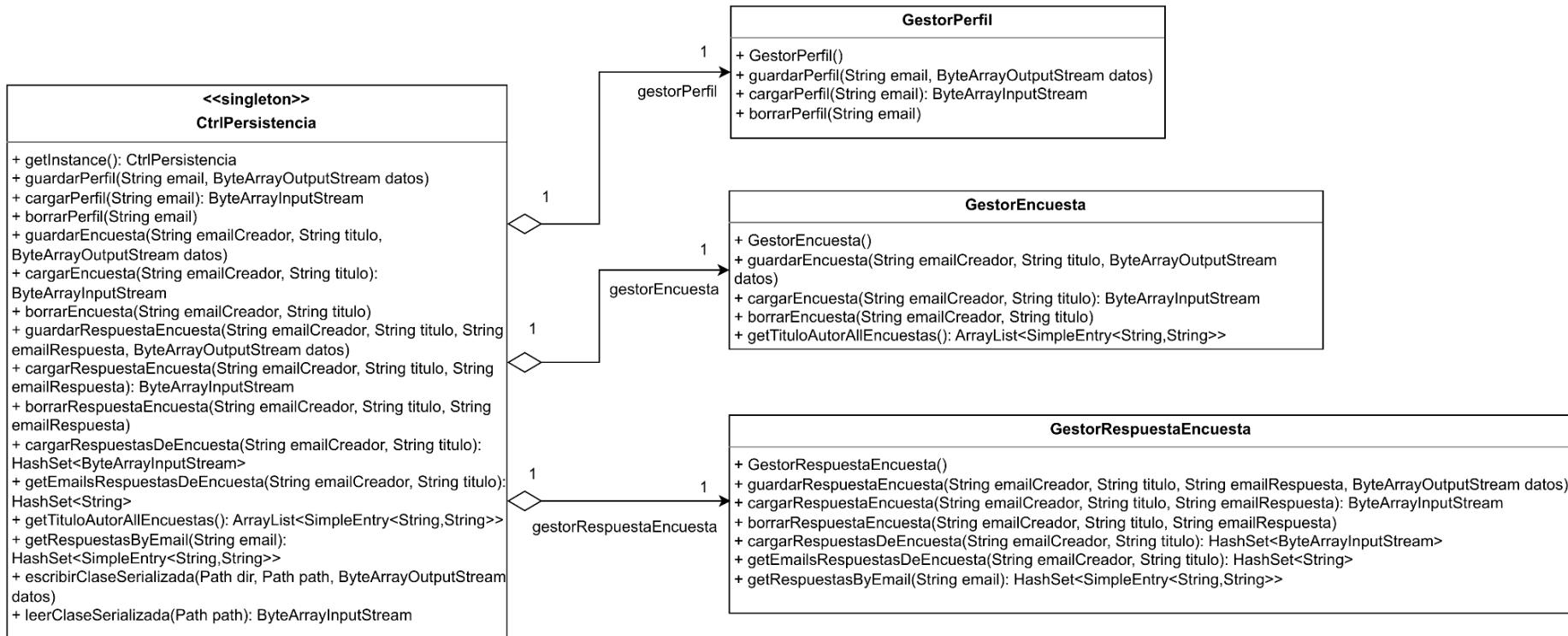
1. Gestión de encuestas: (delegado a CtrlEncuesta)
 - 1.1 Creación de encuestas
 - 1.2 Cargar (título+autor o path)/Guardar/Borrar encuesta
 - 1.3 Consultar encuesta
 - 1.4 Listar por título y autor todas las encuestas que hay en memoria.
 - 1.5 Modificación de una encuesta

- 1.4.1 Añadir preguntas
 - 1.4.2 Eliminar preguntas
 - 1.4.3 Modificar pregunta (tipo, obligatoriedad, texto,...)
 - 1.6 Seleccionar encuesta o pregunta para su respectiva modificación
2. Gestión de Perfiles (delegado a CtrlPerfil)
- 2.1 Crear Perfil
 - 2.2 Cargar un Perfil existente (mail o path)
 - 2.3 Comprobar si existe un Perfil
 - 2.4 Asociar respuestas y encuestas creadas a un Perfil
 - 2.5 Consultar todas las respuestas del perfil cargado
3. Gestión de Respuestas (delegado a CtrlRespuesta)
- 3.1 Crear/Guardar/Cargar(creador+título+respondedor o path)/Borrar una Respuesta
 - 3.2 Añadir los datos de una Respuesta a Pregunta
 - 3.3 Consultar la Respuesta Actual
 - 3.4 Consultar los emails de los autores de las respuestas cargadas
 - 3.5 Consultar la respuesta de un perfil a la encuesta cargada
4. Análisis de Datos (delegado a Analizador)
- 4.1 Elección de algoritmo de clustering
 - 4.2 Elección de Inicializador de algoritmo
 - 4.3 Elección de evaluador de calidad
 - 4.4 Elección del valor k
 - 4.5 Ejecución de análisis
 - 4.6 Obtención de la evaluación
 - 4.7 Obtención de los resultados específicos del análisis

En resumen, se trata de una clase que delega en los controladores especializados y realiza validaciones globales actuando como orquestador de la lógica del dominio.

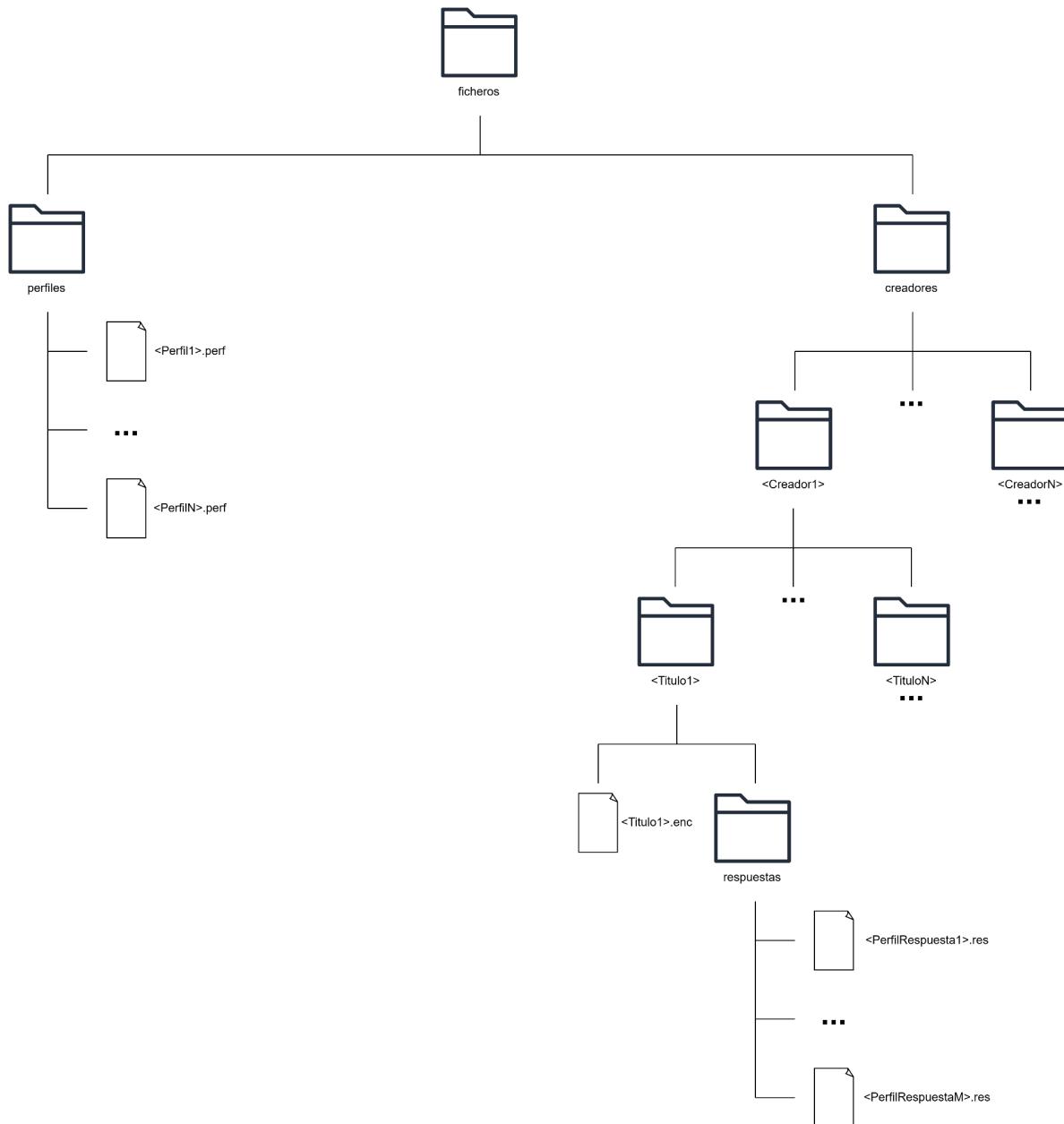
2. Diagrama de la capa de Persistencia

Adjuntamos el diagrama en la carpeta DOCS.



2.1 Estructura de los ficheros

Es importante mencionar también la estructura de los ficheros para entender cómo se guardan en nuestra persistencia. Ilustramos el siguiente esquema para visualizar cómo se estructura:



NOTA: Los ficheros/directorios que no están entre “<>” tienen su nombre explícito como en el diagrama.

La carpeta **ficheros** que contiene toda la estructura se genera automáticamente en tiempo de ejecución mientras se van creando perfiles, encuestas y respuestas, en el mismo directorio de trabajo desde donde se ejecuta el .jar del programa.

En la carpeta de **ficheros** está la carpeta de **profiles** que contiene todos los ficheros de perfiles serializados guardados en la persistencia, con la extensión personalizada de “.perf” y la carpeta de **creadores** que se genera cuando se crea una encuesta por primera vez. Dentro de la carpeta **creadores** tenemos un conjunto de carpetas donde su nombre es el email de los perfiles que han creado encuestas, y dentro de cada carpeta de creador hay un conjunto de carpetas que tienen de nombre los títulos de las encuestas que creó ese creador. Dentro de cada carpeta de título, tenemos un fichero de encuesta serializado con la extensión “.enc” que ha de tener el mismo nombre que la carpeta que pertenece, es decir que tiene de nombre el título de la encuesta, y una carpeta **respuestas** que contiene los ficheros de respuesta serializados con la extensión “.res” que responden a la encuesta de la carpeta anterior. Pueden haber M ficheros de respuesta a una encuesta, donde $M \leq N$ perfiles en el sistema (es decir, que no todos los perfiles han de responder a la misma encuesta, sólo un subconjunto que puede ser vacío).

La estructura se decidió así para agrupar los ficheros de forma que el path que se ha de construir para referenciarlos se deriva de las claves externas definidas en la primera restricción de integridad en la capa de dominio, y así no tener que buscar de forma “arbitraria” y facilitar su búsqueda con una simple línea de código conociendo la estructura y además permitir que puedan haber encuestas con el mismo título siempre y cuando el email del creador sea diferente.

2.2 Descripción de las clases de Persistencia

CtrlPersistencia

La clase CtrlPersistencia es el controlador principal de la persistencia. Usa el patrón singleton para que se pueda referenciar desde cualquier punto del programa, en el caso de nuestro proyecto, se referencia principalmente en los controladores de dominio para guardar, cargar o borrar instancias de las clases que gobiernan los controladores de dominio y también para realizar consultas arbitrarias que se usan

principalmente en la capa de presentación. Delega el trabajo de definir la ruta donde ha de referenciar los archivos de una cierta clase de dominio que se ha de guardar, cargar y borrar a sus gestores, donde usan un pseudo-patrón plantilla para leer y escribir las clases serializadas del propio CtrlPersistencia, ya que la lógica para guardar y cargar es la misma para todas las clases ahorrandonos escribir código repetitivo.

GestorPerfil

La clase GestorPerfil es quien se encarga de guardar, cargar y borrar los ficheros asociados a los perfiles, con la extensión “.perf”. Para guardar usa un ByteArrayOutputStream que contiene el perfil serializado y llama a escribirClaseSerializada de CtrlPersistencia para escribirlo en un FileOutputStream con el path correspondiente según la estructura de los ficheros. Para cargar usa y devuelve el ByteArrayInputStream, generado por el método de leerClaseSerializada del CtrlPersistencia, que contiene el perfil serializado leído desde un fichero de perfil que se deserializa luego en el CtrlPerfil. Y para borrar especifica la ruta e intenta borrar el fichero manualmente desde el gestor, sin usar el CtrlPersistencia.

GestorEncuesta

La clase GestorEncuesta es quien se encarga de guardar, cargar y borrar los ficheros asociados a las encuestas, con la extensión “.enc”. Usa la misma lógica que se detalló anteriormente en el GestorPerfil para guardar, cargar y borrar, esta vez con una encuesta serializada en los ByteArrays que se serializa/deserializa en la capa de dominio, precisamente en el CtrlEncuesta. También tiene una consulta arbitraria que devuelve un conjunto de pares (con la implementación de Java de SimpleEntry) que corresponden al título y autor/creador de todas las encuestas creadas en la persistencia, que se usa luego para mostrar esa información en la capa de presentación.

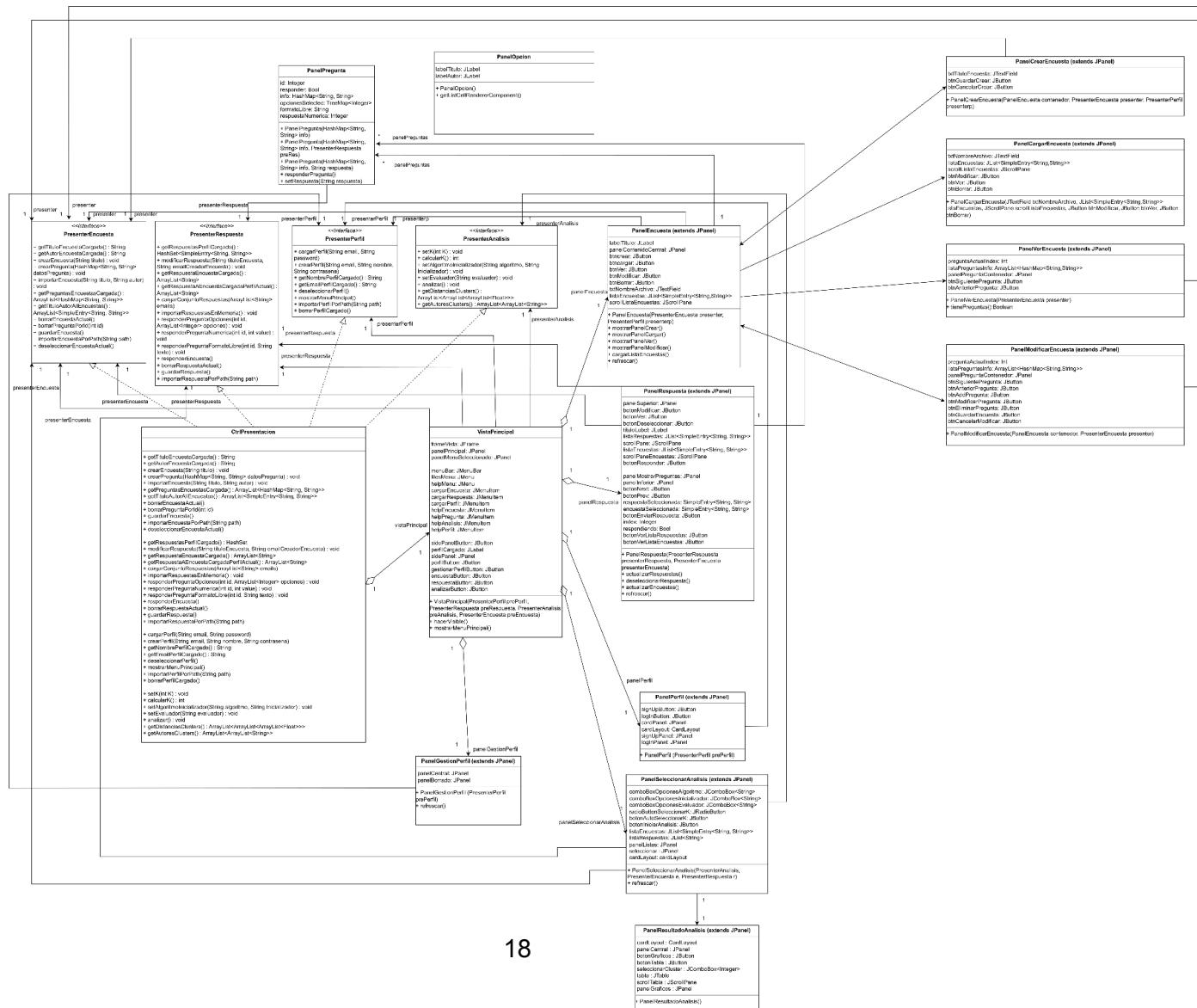
GestorRespuestaEncuesta

La clase GestorRespuestaEncuesta es quien se encarga de guardar, cargar y borrar los ficheros asociados a las respuestas, con la extensión “.res”. Como los otros dos gestores usa la misma lógica de guardar, cargar y borrar, con respuestas que se

serializan/deserializan en el CtrlRespuesta. Pero este gestor además ofrece otra forma de cargar las encuestas filtrando por título y creador de la encuesta en vez de buscar por la clave externa (título, creador y quien responde), para cargar a la vez todas las respuestas de una cierta encuesta para poder realizar de golpe el análisis con esas respuestas.

3. Diagrama de la capa de Presentación

Adjuntamos el diagrama en la carpeta DOCS.



3.1 Descripción de las clases de Presentación

PresenterEncuesta

PresenterEncuesta es una interfaz que contiene las operaciones para consultar, cargar, crear, modificar, borrar y guardar encuestas, además de operaciones que devuelven el título y autor de todas las encuestas del sistema.

PresenterRespuesta

PresenterRespuesta es una interfaz que contiene las operaciones para consultar, cargar, crear, modificar, borrar y guardar respuestas. También permite conseguir el título y autor de todas las encuestas a las que ha respondido el usuario que ha iniciado sesión.

PresenterPerfil

PresenterPerfil es una interfaz que contiene las operaciones para cargar, crear y borrar perfiles. También contiene una operación para mostrar el menú principal, que se usa cuando se haya iniciado sesión.

PresenterAnalisis

PresenterAnalisis es una interfaz que contiene las operaciones para seleccionar los algoritmos y parámetros que se quieren usar para ejecutar el análisis, y consultar los resultados del análisis.

CtrlPresentacion

CtrlPresentacion es el controlador de la capa de presentación. Esta clase implementa todos los presenters, siendo el encargado de hacer llamadas al dominio. De esta forma, solo esta clase tiene acoplamiento con el dominio. Además, gracias a las interfaces que implementa, evitamos que el resto de vistas y paneles tengan conocimiento del controlador y de operaciones que no utilizan en la mayor medida posible. Esta clase crea la VistaPrincipal y el CtrlDominio, inicializando todo el programa.

VistaPrincipal

La clase VistaPrincipal contiene la vista principal del programa y los paneles principales que representan todos los menús de nuestra aplicación. Estos paneles son: PanelRespuesta, PanelPerfil, PanelGestionPerfil, PanelEncuesta, PanelSeleccionarAnalisis y PanelResultadosAnalisis. La vista principal tiene un panel lateral desplegable para navegar por los menús y una barra con opciones para mostrar pequeñas guías sobre cada menú o diálogos para importar archivos externos al sistema que estén en nuestro formato.

PanelRespuesta

PanelRespuesta es una clase que extiende JPanel. Esta clase representa el menú principal de gestión de respuestas. En este menú, se puede seleccionar encuestas para responder y consultar o modificar respuestas del usuario que ha iniciado sesión.

PanelPerfil

PanelPerfil es una clase que extiende JPanel. Esta clase representa el menú de inicio de sesión y de registrarse al sistema. En este menú, el usuario puede iniciar sesión con un perfil que ha creado o crear un nuevo perfil, introduciendo los datos de un perfil y una contraseña.

PanelGestionPerfil

PanelGestionPerfil es una clase que extiende JPanel. Esta clase representa el menú para consultar datos del perfil cargado. También está la funcionalidad de borrar el perfil.

PanelSeleccionarAnalisis

PanelSeleccionarAnalisis es una clase que extiende JPanel. Su propósito es contener los diferentes elementos gráficos que permiten al usuario seleccionar la encuesta y las respuestas de esa encuesta que desea analizar. Además, también permite al usuario decidir el algoritmo de clustering a usar, la inicialización de dicho algoritmo, el número de particiones deseado y la función de evaluación del resultado.

PanelResultadoAnalisis

PanelResultadoAnalisis es una clase que extiende de JPanel. Su propósito es contener los diferentes elementos gráficos que permiten al usuario visualizar los diferentes grupos resultantes del algoritmo de clustering. El panel permite al usuario decidir si la visualización se hace a través de un gráfico o en forma tabular. Además, también deja la opción de visualizar particiones concretas.

PanelEncuesta

PanelEncuesta es el contenedor principal dentro del apartado encuesta del controlador de presentación, y encargado de mostrar los diferentes botones para acceder a los diferentes menús dentro de la gestión de encuestas. Actúa como orquestador de los paneles de funcionalidades específicas (Crear, cargar, Modificar, Ver) y añade la funcionalidad de borrar, esta funcionalidad no requiere de un panel adicional al que delegar la creación del panel ya que es una acción trivial que borra la encuesta cargada en memoria en ese momento, mostrando un mensaje de confirmación. La estructura de este panel se basa en un área superior con los botones de navegación (crear, cargar, modificar, ver y borrar), y un panel con el contenido central para alternar entre las vistas de los diferentes menús según la acción del usuario, la construcción de este panel la delega a 4 paneles hijos distintos (PanelCrearEncuesta, PanelCargarEncuesta, PanelModificarEncuesta, PanelVerEncuesta). Además este panel con tal de mejorar la UX, se encarga de deshabilitar/habilitar botones según si se puede realizar la acción o no y muestra la encuesta seleccionada. Inicialmente, el contenido central contendrá la lista de encuestas de la persistencia (menú de cargar inicial).

PanelCrearEncuesta

PanelCrearEncuesta es un panel cuya responsabilidad es construir un panel en su creadora el cual muestre una interfaz al usuario con las funcionalidades necesarias para que éste pueda crear una encuesta con título y pasar a modificarla para poder añadir las preguntas del tipo y contenido que el usuario elija y pueda cancelar la creación en cualquier momento así como confirmar y que la encuesta quede guardada.

PanelCargarEncuesta

PanelCargarEncuesta es un panel cuya responsabilidad es construir un panel en su creadora el cual muestre una interfaz al usuario con las funcionalidades necesarias para que éste pueda cargar una encuesta y pueda ver en todo momento las encuestas disponibles en la persistencia (título+autor), podrá elegir la encuesta a cargar seleccionándola en la lista de encuestas.

PanelVerEncuesta

PanelVerEncuesta es un panel cuya responsabilidad es construir un panel en su creadora el cual muestre una interfaz al usuario con las funcionalidades necesarias para que éste pueda ver una encuesta navegando entre todas sus preguntas y ver los datos de las preguntas (tipo, texto,...).

PanelModificarEncuesta

PanelModificarEncuesta es un panel cuya responsabilidad es construir un panel en su creadora el cual muestre una interfaz al usuario con las funcionalidades necesarias para que éste pueda modificar una encuesta navegando entre sus preguntas y ofreciendo la posibilidad de modificarlas, crear nuevas o eliminarlas, así como tendrá las opciones de guardar o descartar los cambios.

PanelOpcion

PanelOpcion es una clase reutilizable que se extiende de JPanel e implementa ListCellRenderer. Esta clase se utiliza en varias clases para dibujar las opciones de listas de títulos y autores de encuestas, sustituyendo el ListCellRenderer por defecto de las JList. Este panel nos permite dibujar las opciones de la lista en un formato más personalizado.

PanelPregunta

PanelPregunta es una clase reutilizable que extiende JPanel. Esta clase tiene diferentes formas de crearse, cada una con un propósito diferente. Este panel permite mostrar una pregunta de una encuesta de tres maneras: la primera, para ver la pregunta en modo de consulta, sin opción de responder; la segunda, pasándole la

respuesta para mostrar una respuesta a la pregunta; y la tercera, pasándole el PresenterRespuesta para poder responder a la pregunta. De esta forma, representamos las preguntas de una encuesta en un formato más uniforme para distintas partes de la aplicación.

4. Estructuras de datos y algoritmos usados

4.1. Estructuras de Datos

En este apartado se describen y analizan las estructuras de datos más relevantes que se han usado en el proyecto.

4.1.1 HashSet

Es una estructura que guarda elementos únicos, donde no importa el orden en el que son guardados. Evita duplicidad y se obtiene una búsqueda rápida con complejidad amortizada de $O(1)$ suponiendo que se usa una buena función de hash que no cause demasiadas colisiones (pero que puede crecer a una complejidad de $O(n)$ en el peor caso si la función de hash causa demasiadas colisiones o se produce un *rehash* de la tabla).

Sus usos más relevantes son en:

- **Encuesta:** Utiliza `Set<Pregunta> preguntas = new HashSet<>();`
- **Perfil:** Usa `HashSet<String>` para encuestasCreadas.
- **CtrlDominio:** Usa `HashSet<RespuestaEncuesta>` para cargar las respuestas antes de analizar.

Ejemplos de uso: una Encuesta no puede tener la misma pregunta dos veces, ni un perfil puede tener dos encuestas creadas idénticas.

4.1.2 ArrayList

Es una estructura utilizada para guardar secuencias de elementos donde se hacen accesos por índice o secuenciales de forma frecuente gracias a su complejidad temporal de $O(1)$ amortizado ($O(n)$ en el peor caso donde hay que realoactar el ArrayList para meter nuevos elementos) por acceso por índice que se convierte en $O(n)$ por recorrido, donde n es el número de elementos en el contenedor.

Sus usos más relevantes son los siguientes:

- **ConOpciones**: Se usa `ArrayList<Opcion>` para guardar las opciones de una pregunta.
- **KMedoids y KMeans**: Se usa `ArrayList<RespuestaEncuesta>` para manejar los medoids(centroides) y para los clusters resultantes (`ArrayList<ArrayList<...>>`).
- **RespuestaEncuesta**: Se usa temporalmente en `getDatos()` para devolver los valores en forma de lista.
- **CtrlPresentacion (y CtrlDominio, CtrlEncuesta, CtrlRespuesta)**: Se usa en operaciones que devuelven datos de encuestas o respuestas, donde cada elemento del `ArrayList` representa los datos de una pregunta o la respuesta a una pregunta.

4.1.3 TreeMap

Es una estructura arbórea de tipo diccionario (*Key -> Value*) que mantiene las *Keys* ordenadas según el criterio de ordenación que establece la clase de *Key* a no ser que se especifique lo contrario en la constructora. Además, las operaciones de búsqueda, inserción y borrado tienen complejidad temporal asegurada de $O(\log n)$, donde n es el número de elementos en el contenedor.

Sus usos más relevantes son en:

- **RespuestaEncuesta**: Usa `TreeMap<Integer, TDatos>` `idRespuesta` para mantener los datos correspondientes a las respuestas de cada pregunta individual del cuestionario siempre ordenados.
- **TDatosOpciones**: Se usa `TreeMap<Integer, Integer>` en el cálculo de la moda en los conjuntos de opciones. Más concretamente, a la hora de contar las frecuencias, es muy útil poder acceder al contador de una opción en tiempo $O(\log n)$. Se ha optado a usar `TreeMap` en vez de `HashMap` a pesar de un mayor tiempo de búsqueda para poder asegurar una resolución clara en caso de empate entre diferentes opciones con la misma frecuencia.

4.1.4 SimpleEntry

Es una estructura que guarda un par de valores relacionados, actuando como una implementación muy simple de un *Pair*.

Sus usos más relevantes se destacan en:

- **Perfil:** Usa `Set<SimpleEntry<String, String>>` `respuestasHechas` donde relacionamos el Key como el título de la encuesta y el Value como el creador de la encuesta.
- **PanelEncuesta, PanelOpcion, PanelCargarEncuesta, PanelCrearEncuesta, PanelSeleccionarAnalisis:** Para usarlo como dato primitivo para mostrar y manipular encuestas, donde Key es título de la encuesta y el Value el creador de la encuesta, y desacoplar el dominio de la capa de presentación.
- **PanelRespuesta:** Almacenar las claves que identifican las encuestas las cuales el perfil actual cargado ha respondido, adquiridos por el método `getRespuestasPerfilCargado()`, donde Key es título de la encuesta y el Value el creador de la encuesta.
- **CtrlPresentacion:** Como argumento en la implementación dos métodos de los presenters que se usan en los paneles, `getTituloAutorAllEncuestas()` y `getRespuestasPerfilCargado()`, que en este controlador son wrappers para llamadas a dominio vía el `CtrlDominio`.
- **CtrlDominio:** Para los dos métodos mencionados anteriormente (`getTituloAutorAllEncuestas()` y `getRespuestasPerfilCargado()`), actua como puente delegando la responsabilidad al `CtrlPerfil` y `CtrlEncuesta` que entorno llaman a la persistencia que devuelve el resultado final.
- **CtrlPersistencia:** Implementa las consultas arbitrarias de `getTituloAutorAllEncuestas()` y `getRespuestasPerfilCargado()`, almacenando en el Key el título de la encuesta y en el Value el email del creador de la encuesta.

4.1.5 HashMap

Es una estructura de tipo diccionario (key-value) que permite guardar pares de elementos con este formato. Al igual que HashSet, utiliza la función hash para determinar dónde almacenar cada par para proporcionar una complejidad temporal O(1) o O(n) en operaciones de búsqueda, inserción o eliminación en caso peor con una función pobre de Hash que cause colisiones. Su orden no es predecible ni está garantizado y requiere que las claves sean inmutables para mantener la eficiencia O(1)

Sus usos relevantes son:

- **CtrlPresentacion** (y **CtrlDominio**, **PanelEncuesta**,...) en crearPregunta() y getPreguntasEncuesta() se usa `HashMap<String, String>` para manejar la información de las preguntas como un tipo básico construyendo el HashMap a través de `key("NombreAtributo")->"Valor"`.

4.2 Algoritmos

4.2.1 K-Means

El algoritmo de K-Means consiste en particionar el conjunto de respuestas de la encuesta en un número de grupos (clusters) de manera que se minimice la varianza interna de los clusters. Para este propósito, usa entidades matemáticas llamadas centroides. En nuestro caso, un centroide de un cluster se puede definir como la media de las respuestas pertenecientes a la partición.

En este caso, la varianza se define como el sumatorio de las distancias al cuadrado entre los puntos de un cluster a su centroide. Algunas de las ventajas de usar esta métrica son: evitar calcular raíces cuadradas y el hecho de minimizar la suma de distancias cuadradas implica la minimización de la suma de distancias.

El problema a resolver se ha comprobado que está en la clase NP-Hard en general, por lo que se usan algoritmos heurísticos como el de Lloyd que usan búsqueda local en vez de fuerza bruta. No obstante, se ha podido comprobar experimentalmente que estas implementaciones siguen teniendo una complejidad superpolinómica, más concretamente, la complejidad temporal teórica del algoritmo de Lloyd es $O(n \cdot k \cdot d \cdot i)$.

Donde n es el número de respuestas, k es el número de clusters, es decir el número de particiones, d es la dimensionalidad de cada respuesta, es decir, cuántas respuestas a preguntas individuales tiene esa respuesta, e i es el número de iteraciones hasta que el algoritmo converge, que sabemos que es $2^{\Omega(\sqrt{n})}$.

El propio algoritmo de Lloyd consiste en inicializar los centroides para cada grupo, asignar cada respuesta al cluster cuyo centroide sea el más cercano a la respuesta, donde la noción de distancia es la distancia cuadrada, recalcular los centroides para cada partición y repetir este proceso hasta que los centroides converjan (en este caso cuando los centroides recalculados sean idénticos a los centroides originales).

Cabe destacar que en nuestra implementación admitimos la inclusión de agrupaciones vacías dentro de la solución final. Esto puede ser causado si se usa inicialización aleatoria de los centroides iniciales y 2 de ellos son respuestas idénticas. Consideramos que esta situación es suficientemente rara, sobre todo a medida que crece n , que en esta implementación básica no hace falta tratarla de forma excepcional.

Las responsabilidades de generación de centroides iniciales, cálculo de distancias y recálculo de centroides se dejan a clases externas a la clase que implementa el propio algoritmo. Por lo que la implementación simplemente sigue el procedimiento explicado anteriormente.

Una vez entendemos cómo funciona el algoritmo, podemos analizar su coste temporal en nuestra implementación y compararla con el coste teórico.

Primero de todo, el algoritmo usa un bucle *while* con condición de convergencia, por lo que el número de iteraciones es $2^{\Omega(\sqrt{n})}$.

En segundo lugar, inicializamos un *ArrayList* de *ArrayLists* que guarda la solución. Esto tiene un coste temporal en peor caso de $O(k)$, porque simplemente se hacen k llamadas a la constructora por defecto de *ArrayList* que tiene coste $O(1)$.

A continuación, para cada respuesta, recorremos todos los centroides de esa iteración para encontrar el más cercano. Como se puede intuir, esta acción tiene un coste temporal de $O(n \cdot k \cdot \text{distancia})$ al ser implementado como un *for loop* anidado.

Más concretamente, distancia se refiere al coste de calcular la distancia entre 2 respuestas. En nuestro caso, la complejidad temporal varía dependiendo del tipo de dato con el que se esté trabajando. Sin embargo, podemos colocar una cota superior al coste teniendo en cuenta que el cálculo de distancias para preguntas de tipo formato libre tiene el peor coste.

A causa del hecho de que usamos distancia de Levenshtein para calcular la distancia entre 2 textos, el coste del cálculo, siguiendo una implementación de programación dinámica en estilo tabulación, es de $O(l_1 \cdot l_2)$, donde l_1 es la longitud del texto 1 y l_2 es la longitud del texto 2. Por lo que la cota superior es $O(l_1 \cdot l_2 \cdot d)$ teniendo en cuenta que en el peor caso, todas las preguntas individuales de una encuesta son de formato libre.

Seguidamente, el recálculo de los centroides implica recorrer los centroides antiguos y reemplazarlos por nuevos. El coste del recálculo es de $O(n \cdot d \cdot l \cdot \log(d \cdot l))$ en peor caso, donde l es la longitud promedio de palabras de los textos de las preguntas de formato libre presentes en el cluster.

Esto se debe a que, como cota superior, todas las respuestas están en el mismo cluster y que el cálculo de una componente de centroide para preguntas de formato libre requiere operaciones sobre un TreeMap dentro de un *for loop* anidado sobre las dimensiones de la respuesta (respuestas a preguntas individuales) y sobre las palabras del texto en caso de que sea una pregunta de tipo formato libre.

Finalmente, la comprobación de convergencia es simplemente $O(k)$, ya que solo se mira si coinciden los valores de los centroides.

Por último, destacamos la posibilidad de mejorar esta implementación a través de como precálculo de distancias, uso de cache o explotar la propiedad de desigualdad triangular de las distancias (no cuadradas) para ahorrar cómputo.

Como conclusión, nuestra complejidad temporal, que aproximamos como $O(i \cdot n \cdot d \cdot l \cdot \log(d \cdot l))$, se acerca bastante a la teórica, aunque en nuestro caso el coste de recalcular centroides domina sobre los cálculos de distancias.

4.2.3 K-Means-Optimizado

La idea principal en la que se basa esta optimización, como se ha insinuado anteriormente, es explotar la propiedad de desigualdad triangular que cumplen nuestras métricas de distancia para calcular *upper bounds* y *lower bounds* que nos ayuden a evitar calcular distancias.

Más concretamente, usamos 2 lemas que se pueden deducir de ella¹. En primer lugar, sea x un punto y b, c centroides, $d(b, c) > 2 \cdot d(x, b)$ implica $d(x, c) \geq d(x, b)$. Por lo que si se cumple el antecedente, nos podemos ahorrar calcular la distancia $d(x, c)$. En segundo lugar, $d(x, c) \geq \max(0, d(x, b) - d(b, c))$. Este segundo lema nos ayudará a recalcular los *bounds* de manera más eficiente.

El algoritmo como tal sigue una estructura muy parecida a la del K-Means ordinario, por tanto, esta explicación solamente recalcará las diferencias fundamentales entre los 2 algoritmos y se aprovechará para argumentar la complejidad temporal en su lugar.

Primeramente, se inicializan las matrices y vectores que se usarán para información adicional como *lower bounds*, *upper bounds*, matrices de distancias precalculadas, matrices que indican validez de ciertos *bounds*, entre otros. En general, podemos concluir que el coste de todas estas operaciones de inicialización es dominado por el precálculo de distancias entre centroides, con un coste de $O(c^2 \cdot distancia)$.

Notamos que ya se ha discutido anteriormente el coste de calcular una distancia.

¹ La demostración formal se puede encontrar en <https://cdn.aaai.org/ICML/2003/ICML03-022.pdf>

Seguidamente, el cálculo inicial de los *bounds* viene dado por una función de inicialización que recorre, por cada respuesta, todos los centroides. Por una justificación similar a la hecha en K-Means, esto tiene un coste de $O(n \cdot k \cdot \text{distancia})$. Cabe destacar que, en principio, ya intentamos aplicar el lema 1 con la intención de ahorrar algunos cálculos, sin embargo, suponiendo peor caso esto no se da.

La comprobación de validez posterior simplemente recorre todos los *upper bounds*, lo cual tiene un coste de $O(n)$.

Al entrar en el bucle principal, recalculamos todas las distancias necesarias según el vector de validez. En general, podremos ahorrarnos gran parte de estos cálculos. No obstante, en peor caso, suponiendo que no podemos, esta parte también tiene un coste $O(n \cdot k \cdot \text{distancia})$ al igual que K-Means.

El recálculo de centroides es idéntico al de K-Means, por lo que tiene coste $O(n \cdot d \cdot l \cdot \log(d \cdot l))$.

A continuación, recalculamos las matrices auxiliares según estos centroides nuevos. De nuevo, el coste es dominado por el precálculo de distancias entre centroides.

Finalmente, tenemos la comprobación de convergencia que es al igual que en K-Means, $O(k)$.

En conclusión, en caso peor la complejidad temporal es la misma que K-Means. Aunque experimentalmente se puede apreciar un *speedup* considerable.

4.2.2 Inicialización Random

Los métodos de inicialización de centroides son muy importantes a la hora de determinar su velocidad de convergencia. Una inicialización buena permite reducir el factor exponencial del número de iteraciones del algoritmo de forma considerable.

Una implementación trivial es simplemente seleccionar de forma aleatoria las respuestas del conjunto para que actúen como centroides iniciales. Como se ha

descrito anteriormente, esto conlleva ciertos inconvenientes en casos donde se seleccionan 2 respuestas idénticas como centroides. Aparte de eso, cabe destacar que es crucial usar un generador de números aleatorios que siga una distribución uniforme para acelerar la convergencia.

En nuestra implementación, simplemente se generan números aleatorios usando una distribución uniforme entre 0 (inclusivo) y k (exclusivo) de manera que no se repita ninguno (se asegura usando un HashSet). A continuación, se devuelven los centroides que son las respuestas cuyo índice coincide con el del número generado. Por lo que la complejidad temporal es $O(k)$, siendo k el número de clusters deseados.

4.2.3 K-Means++

K-Means++ es otro método de inicialización que se especializa en encontrar centroides iniciales decentemente buenos para el algoritmo de K-Means de forma voraz. Más concretamente, el algoritmo asegura que la solución de clustering resultante después de usar K-Means++ es $O(\log k)$ competitivo con la solución óptima global.

Este algoritmo soluciona un problema bastante importante de la inicialización aleatoria que es su pobre fiabilidad, ya que se pueden generar centroides arbitrariamente malos, que extiendan innecesariamente el tiempo de convergencia y/o produzcan clusterings lejanos al óptimo. Además, si se añade alguna componente estocástica a la implementación de K-Means++, entonces se puede argumentar que tampoco se pierde mucho acceso a las regiones del espacio de soluciones que nos interesan.

K-Means++ consiste en primeramente elegir un centroide de forma aleatoria siguiendo una distribución uniforme, en segundo lugar, calcular las distancias cuadradas de cada respuesta a ese centroide, en tercer lugar, elegir un nuevo centroide de forma aleatoria pero siguiendo una probabilidad con pesos proporcionales a la distancia cuadrada calculada anteriormente y, finalmente, repetir los pasos 2 y 3 hasta elegir los centroides.

Como se puede deducir de la explicación anterior, este algoritmo intenta asegurar que los centroides estén relativamente separados entre ellos, lo cual acelera el proceso de convergencia del algoritmo K-Means. Además es interesante destacar que este método asegura que se escoge la misma respuesta como centroide porque la distancia entre la respuesta y el centroide es 0, por lo que en consecuencia, la probabilidad de escoger esa respuesta también es 0.

En nuestra implementación, cabe destacar que la manera en la que se implementa la elección de centroide con probabilidad proporcional a la distancia cuadrada es mediante definir un umbral aleatorio entre 0 (inclusivo) y el sumatorio de distancias cuadradas total (exclusivo). Entonces, se escoge como centroide la primera respuesta cuya contribución a la suma acumulada de distancias cuadradas supere ese umbral.

Es importante destacar que la implementación se podría optimizar de diversas maneras como por ejemplo manteniendo una caché o simplemente precalculando distancias.

4.2.4 K-Medoids

El problema de k-Medoids clustering, que tiene la misma finalidad que el k-Means de particionar respuestas en clusters, es clasificado como un problema NP-difícil, por tanto, la implementación que usamos para resolverlo aplica ciertas heurísticas para que se pueda ejecutar el algoritmo en tiempo polinómico de forma *greedy*, que es más rápido que realizar una resolución exhaustiva, a pesar de que no resuelve el problema con los medoides óptimos para realizar el clustering.

El algoritmo empleado para resolver el clustering es el *Partitioning Around Medoids* (PAM), inventado por Leonard Kaufman y Peter J. Rousseeuw. Consiste en dos fases: una fase build que construye los medoides, del cual se encarga la inicialización Greedy (explicada más adelante), y la fase swap, cuyo funcionamiento es el siguiente, descrito en pseudocódigo:

dado: $X = \{x_1, \dots, x_n\}$ puntos de datos,

$k \in \mathbb{N}$,

$M = \{Y \subseteq X \mid Y = \text{greedy}(X, k)\}$,

$\text{coste}_{\text{total}}(M, X)$:

$c := 0$

$\forall x \in X$:

$\text{dist}_{\min} := +\infty$

$\forall m \in M$:

$\text{dist} := \sum_{k=1}^d |x_k - m_k|$ (distancia entre x y m)

si ($\text{dist} < \text{dist}_{\min}$):

$\text{dist}_{\min} := \text{dist}$

$c := c + \text{dist}_{\min}$

devolver c

$PAM_{SWAP}(M, X, k)$:

$\text{converge} := \text{false}$

$\text{coste}_{\text{mejor}} := \text{coste}_{\text{total}}(M, X)$

while($\neg \text{converge}$):

$\text{converge} := \text{true}$

$\forall x \in X$:

$\forall m \in M \mid m \neq x$:

$M := \{M \mid x \in M \wedge m \notin M\}$ (swap)

$\text{coste} := \text{coste}_{\text{total}}(M, X)$

si ($\text{coste} < \text{coste}_{\text{mejor}}$):

$\text{coste}_{\text{mejor}} := \text{coste}$

$\text{converge} := \text{false}$

sinó:

$M := \{M \mid x \notin M \wedge m \in M\}$ (deshacer swap)

devolver M

El resumen básico del pseudocódigo es que calculamos el coste total, que es la distancia mínima total de cada respuesta con cada medoid del conjunto obtenido del inicializador Greedy; y luego, para cada respuesta que no sea un medoid, vamos intercambiándolo en el conjunto de medoids con cada medoid y recalculamos el coste para mejorar si es menor que el mejor coste total calculado hasta ahora; si el coste no mejora, quiere decir que el algoritmo ha convergido y el coste no se puede mejorar, devolviendo el conjunto de medoids en el cual luego con la función *construirClusters* vamos construyendo el conjunto de clusters, donde cada respuesta se la asigna al cluster que pertenece el medoide más cercano. El coste temporal de esta implementación básica del algoritmo, donde recalculamos todo el rato el coste total, és de $O(n^2k^2)$, donde n es el número de elementos en el conjunto de respuesta.

Para la nueva segunda versión del algoritmo, se realiza una optimización al k-Medoids cambiando la implementación básica por la implementación de *FasterPAM*, que reduce la complejidad temporal general de $O(n^2k^2)$ a $O(nk)$ en caso medio ($O(n^2k^2)$ en el peor caso, el mismo coste que la implementación básica pero en casos prácticos se da pocas veces). Ahora, en vez de recalcular el coste total cada vez que hacemos un intercambio, tenemos 3 arrays auxiliares: uno que contiene los índices de los medoids más cercanos y dos que contienen las distancias y las segundas distancias más cercanas. Cada índice de cada array auxiliar corresponde al índice de ArrayList de respuestas que analizamos, es decir, que dado una respuesta de índice i : la posición i de cada array auxiliar referencia información a esa respuesta.

La explicación general de la optimización es, en vez de recalcular el coste todo el rato (que es costoso), usamos los arrays de distancia y segunda distancia más cercana como caches para guardar las distancias que se usan, cada vez que haya una mejora actualizamos la distancia más cercana y la segunda distancia más cercana la sobreescrivimos con el viejo valor de distancia más cercana, y así reemplazamos el tener que recalcular los costes con recomputar los caches de distancias si hubo una mejora en las distancias (que es más rápido).

4.2.5 Inicialización Greedy

El algoritmo Inicializador Greedy construye el conjunto inicial de medoids para el algoritmo *k-medoids* explicado antes, seleccionando el primer medoid de forma aleatoria e iterando hasta completar los medoids necesarios definidos, para cada respuesta candidata que no sea medoid se evalúa el resultado de incorporarla como medoid al conjunto actual. Esta evaluación consiste en calcular para cada respuesta del dataset la distancia mínima entre dicha respuesta y el conjunto de medoids con el medoid candidato. El medoid se selecciona si minimiza la suma total de las distancias, repitiendo este método hasta conseguir los medoids necesarios definidos previamente (el número de *clusters/agrupaciones*).

El método actúa como una heurística determinista (salvo la inicialización aleatoria) para aproximar una buena configuración inicial para el *k-medoids*.

4.2.6 Silhouette

El coeficiente de Silhouette es un método de interpretación y validación de la coherencia dentro del análisis de grupos.

El valor de Silhouette mide cuán similar es un punto a su propio cluster en comparación con otros clusters. Este valor va de -1 a +1, donde un valor más alto indica un mejor resultado.

Para cada punto i, Silhouette compara:

- a(i): la cohesión, es decir, la distancia promedio del punto a los demás puntos de su mismo clúster.
- b(i): la separación, o distancia promedio del punto al clúster vecino más cercano (aquel que minimiza esta distancia).
- Entonces, el coeficiente de Silhouette de un punto es $s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$

El coeficiente global será la media de todos los s(i).

En el caso peor, suponiendo que todas las preguntas son de FormatoLibre, teniendo en cuenta que para cada punto se calcula su distancia respecto a todos los demás puntos, que solo hacemos accesos a ArrayList (coste temporal constante) y que el

resto de operaciones (min, max, etc.) son constantes, el algoritmo tiene un coste de $O(n^2d)$, donde d es la dimensionalidad de las respuestas, mencionado anteriormente en KMeans.

Aunque exista el Silhouette simplificado con un coste menor, nosotros hemos decidido implementar el completo.

4.2.7 Calinski-Harabasz

El índice de Calinski-Harabasz compara la separación entre clusters (varianza entre grupos, BCSS) y la compactad dentro de esos grupos (varianza interna, WCSS).

Esos valores se calculan así:

$$BCSS = \sum_{i=1}^k n_i \|c_i - c\|^2$$

$$WCSS = \sum_{i=1}^k \sum_{x \in C_i} \|x - c_i\|^2$$

La fórmula queda como $CH = \frac{BCSS/(k-1)}{WCSS/(n-k)}$, donde CH es mejor cuanto más alto sea su valor.

4.2.8 Davies-Bouldin

El índice de Davies-Bouldin mide cuán bien separados y cuán compactos son los clusters. Este índice indica un mejor clustering cuanto más cerca del 0 sea el resultado.

Para calcularlo, nosotros hemos hecho:

$$DB = \frac{1}{k} \sum_{i=1}^k R_i$$

donde:

$$R_i = \max_{j \neq i} R_{ij} \text{ y } R_{ij} = \frac{S_i + S_j}{M_{ij}}$$