Bases de Datos no Relaccionales

Miguel Esteban, Carlos Boned, Arnau Puche

Resumen—En este documento explicaremos todo el proceso que hemos seguido para completar el proyecto de la asignatura, desde la configuración de la base de datos, pasando por la insercción de los *datasets*, así como la generación de resultados experimentales.

Index Terms—MongoDB, Python, Research, Neural Network, Machine Learning, Data Classification, Outliers Detection.

INTRODUCCIÓN

Se nos ha propuesto diseñar y desplegar una base de datos que almacene diferentes *datasets* para que un grupo de investigación pueda utilizarlos en sus investigaciones y registrar los resultados de sus experimentos de manera fiable. Para llevar a cabo este trabajo, utilizaremos una base de datos no relacional, por su **flexibilidad** a la hora de almacenar los *datasets* tan diversos que utilizarán:

- Iris Data Set [1].
- Ionosphere Data Set [2].
- Breast Cancer Wisconsin Data [3].
- Letter Recognition Data Set [4].
- The MIR Flickr Retrieval Evaluation [5].

Todos ellos contienen un valor en común, «*label*», que determina conocer la clase a la que pertenece el conjunto de datos; para los cuatro primeros es un *vector de características* y para el último, una imagen `jpg`.

Para el despliege de la base de datos disponemos de una LAN privada formada por 9 servidores MongoDB conectada a Internet y todos los archivos referenciados están disponibles en el repositorio del proyecto [6].

1. INSERCIÓN DE LOS DATOS

Antes de poder realizar cualquier experimento, el equipo debe tener a su disposición los *datasets*. Nuestra propuesta consiste almacenar cada uno de estos en una **colección**, para que puedan ser accedidos de manera rápida y simple[1]. En MongoDB la información se almacena en **documentos** de formato `BSON` con un tamaño máximo de 16 MB [7]; estos documentos están agrupados en colecciones dentro de una base de datos [8]. El archivo encargado de esta tarea es `insertData.py`, a continuación se analizarán los aspectos más importantes de este fichero.

1.1. existDataset

Esta función comprueba si el *dataset* que se quiere insertar ya está disponible en la base de datos. La comprobación que realiza se basa en **comparar el número de elementos** de la colección del mismo nombre con el número de filas del archivo a insertar. Si dichos números coinciden, se informa al usuario de que el *dataset* ya está disponible, si por el contrario esta comparación no es positiva, se eliminará la colección, si existía; y se cargará la información del archivo.

1. Consultar el Anexo para conocer las distribuciones específicas.

1.2. deleteDataset

En caso de que el *dataset* no esté completo, es necesario **eliminar la colección** para permitir insertar nuevamente el fichero. Esta tarea se realiza mediante un `db.collection.drop` de la colección indicada. Nos decantamos por esta función, aunque implique aumentar los privilegios del gestor de datos, ya que de si se quiere eliminar algún dato en concreto se deben realizan comparaciones hasta encontrar dicho elemento y posteriormente recorrer el *dataset* para insertarlo correctamente.

1.3. Inserción de los datos

Como se requieren **dos formatos diferentes** de datos (*vector de características*, imágenes), se ha implementado una función para cada tipo. Las imágenes tienen información añadida, ya que se nos requirió también almacenar información relativa a cada red neuronal (*descriptores*) para el correcto funcionamiento de estas.

1.3.1. insertVectorDataset

Este método permite añadir los diferentes *datasets* obtenidos del *UC Irvine Machine Learning Repository*, que contienen la información en archivos `csv`. Para las colecciones se utilizan los nombres aportados por terminal al inicio de la ejecución; es necesario también añadir un archivo con los metadatos[2] del archivo. Estos datos complementarios permiten clasificar los campos de manera correcta y aseguran una correcta inserción de los datos.

La **implementación** en python consiste en la lectura linea a línea del archivo, limitando los diferentes campos mediante el caracter propio de los archivos `csv`: «,». Una vez separada la *label* del vector, se comprueba si existe un campo específico de identificación para cada elemento, en caso afirmativo (*Breast Cancer Wisconsin*) se añade junto a la *label*, el vector característico y junto con un campo extra, `utility`, cuya función se especificará más adelante.

Listing 1. Formato query vector

```
"label":label, "vector":[], "utility":"Outliers"
```

2. Este `JSON` debe indicar la posición del *label* y el número de campos que conforman el vector. Para más información, consultar el Anexo.

1.3.2. *insertImageDataset*

La inserción de la información del *dataset* **MirFlickr 25K** se realiza de manera diferente. Con objeto de no sobrecargar la base de datos, se almacenará la ruta para acceder a cada imagen, que estará disponible en el mismo *clúster*, en un directorio diferenciado. Este *dataset*, almacena las *labels* de manera diferente, mediante archivos nombrados `<label>.txt` que contienen las `IDs` que se corresponden a cada *label*.

La **implementación** de esta función consiste en la creación de un diccionario de la forma {`ID:[labels]`}, que permite añadir posteriormente los *descriptores* en caso de que fuesen necesarios. Mediante este método, podemos asignar el identificador interno de MongoDB, `_id`, al número correspondiente a cada imagen. De esta manera, en un primer momento y sin *descriptores*, la información que se registraría sería:

Listing 2. Formato query imágenes

```
{"_id":img, "label":label[img], "path":~/imgXX.jpg
```

1.3.3. *insertDescriptor*

Esta función únicamente es llamada desde la anterior, `insertImageDataset`. Se encarga de añadir a cada imagen toda la **información extraida** por cada red neuronal. Estos datos se encuentran almacenados en archivos binarios que deben ser leidos y codificados correctamente para poder ser utilizados. Dos estructuras diferentes son usadas:

LABEL — El vector correspondiente a cada imagen forma una matriz ($25000 \times x$), con x número de características. Indican a que clases corresponde cada imagen.

S & V — Ambos ficheros, *Sigmoid* y *Visual*, forman un vector característico con información de cada red neuronal, tienen longitud variable dependiendo de la red. Son idénticos para todas las imágenes.

Esta información está almacenada en un diccionario con la estructura {`NN_X:[]`}, donde *NN* corresponde a la red y *X* al descriptor. Estos diccionarios se añaden en con una nueva clave al creado anteriormente durante la insercción de las imágenes; por lo que antes de realizar las insercciones, la información se encuentra almacenada en un diccionario con la forma siguiente:

Listing 3. Formato query imágenes y descriptores

```
{"_id":img, "label":label[img], "descriptors":{},
    "path":~/images/imgXX.jpg
```

Una vez realizado este paso, en caso de que dichos descriptores existan, se procederá a la insercción de toda la información siguiendo los parámetros indicados en los *listings* 2 y 3.

2. SEGURIDAD. GESTIÓN DE USUARIOS

En el apartado de seguridad se nos pedía implementar el *flag* correspondiente a las **autorizaciones** de entrada con usuarios, así como la creación de **3 usuarios** con roles cada vez más restrictivos. En la carpeta `Configuraciones`, donde se encuentran las diferentes configuraciones usadas durante las sesiones. `standAlone.conf` ha sido la utilizada en las primeras sesiones como la configuración básica con las *flags* necesarias por el momento. Antes de realizar esta configuración teníamos que crear un **super usuario** con privilegios de root y distribuirlo a los diferentes mongos. Para acelerar la creación utilizamos un script el cual detallaremos más adelante. Una vez creado el usuario, se crearon una serie de usuarios con privilegios específicos.

- **GESTORDADES**. Encargado de **administrar** los datos que se deben insertar. Tiene privilegios con operaciones CRUD[3]. Para proporcionar estos privilegios, hemos realizado y otorgado un *rol* llamado `CRUD`. Este usuario únicamente puede realizar estas funciones en la base de datos `Outliers`. Con este usuario hemos insertado todas las colecciones debido a la manera en la que hemos implementado el proyecto, ya que puede realizar actualizaciones y eliminaciones.
- **TEST**. Usuario con **menos privilegios**, únicamente puede leer y escribir, no puede realizar eliminaciones ni actualizaciones en la base de datos. Para este usuario también hemos creado un *rol* específico, `BASIC`.
- **SYS**. Posee **todos los poderes** sobre todas las bases de datos, tanto de las bases de datos locales como de configuración. Este usuario ha realizado todas las configuraciones necesarias en la base de datos.

Para entrar con los diferentes usuarios hemos realizado una serie de contraseñas para cada uno. Para el usuario `sys` la contraseña es *sys*; para el usuario `GestorDades`, *gestor* y para el usuario `Test`, *test*. La validación de los diferentes permisos de los usuarios se realiza mediante un script, «*validacio*», que muestra las diferentes acciones que pueden realizar los usuarios. Más adelante, en la sección scripts, hablaremos de el funcionamiento del mismo.

3. PROTECCIÓN DE LOS DATOS. *ReplicaSet*

En una base de datos distribuida es importante tener **medidas de protección** para poder recuperar los datos en caso de fallos del servicio o humanos. Realizar una **replicación** de datos a los diferentes *clústers* es una manera de evitar la pérdida de información. Para realizar una replicación de los datos hemos utilizado una serie de *scripts*, así como diferentes ficheros de configuración para el *clúster* `main` y los `mongos` del 1 al 6. Durante este proceso hemos generado un *script* `Replicación` que se encarga de automatizar el *shutdown* del servidor mongod y el *boot* con la configuración `Master-Slave`. Este fichero ha ido sufriendo cambios sutiles para las diferentes situaciones; se nos solicitaba un *ReplicaSet* de 3 mongos. El fichero de las máquinas `main`, `mongo-1` y `mongo-2` tienen el nombre `rs0`. Para las máquinas `mongo-3`, `mongo-4` y `mongo-5` existe la misma configuración con nombre `rs1`. Más adelante la `main` tendrá una configuración diferente que será debidamente detallada.

3. Lectura, escritura, insertar, actualizar, eliminar y sus derivados.

4. GESTIÓN DE DATOS. *Sharding*

Un *shard* es una **división de colecciones** con una cantidad muy elevada de datos para no sobrecargar el servidor y que en caso de error se pueda recuperar la información. Para implementarlo tenemos que acceder a los *replicasets* ya creados y activar las `flag` correspondientes. Una vez realizados los pasos indicados por la documentación de MongoDB [9], activamos el enrutador de mongodb, *mongos*. Hemos generado un fichero de configuración para los `mongos-6`, `mongos-7` y `mongos-8` que actuarán como *clústers* de configuración. Este fichero se encuentra en la carpeta `Configuraciones` de nuestro repositorio con el nombre *Configsvr.conf*. Una vez desplegados, iniciamos el `mongos` de tal manera que el cluster `main` utilize el puerto de escucha `27017`. Para evitar problemas con los puestos de escucha, modificamos el puerto al `27018`. Con este cambio se deben modificar las configuraciones del replica set `rs0` para que el `main` esté en el puerto correspondiente. Por otra parte, ajustaremos la configuración de `mongod` para que escuche por el puerto correspondiente. Al ser una configuración especial, esta se encuentra en la carpeta de `configuraciones` con el nombre *Master-Slave-18.conf*. Una vez realizados todos estos cambios, con el `mongos` configurado correctamente y los *shards* configurados se puede dividir las colecciones. Las que hemos seleccionado son *MirFlickr25K* y *Letter-Recognition*.

4.1. *Sharding* MirFLickr25K

Para dividir esta colección hemos creado una *shared key* para que al diversificar la información, esta esté balanceada y los `chunks` generados sean equitativos. Elegir una *shared key* es muy circunstancial, ya que depende de los tipos de query que se quieran realicen. En nuestro diseño esta colección tiene una id incremental, por lo tanto no podemos generarla con un índice de ordenación sobre la `id`. Generar un índice sobre un `array` de strings no está permitido por la implementaciones de MongoDB, por tanto nos hemos decantado por una ***Hashed Shared Key***. Con esta selección evitamos el *hot spot* producido al utilizar índices incrementales como clave; por lo que un *hashing* del índice és más eficiente y elegante.

4.2. *Sharding* Letter-Recognition

Para el dataset *Letter-Recognition* hemos hemos seguido la misma dinámica de la colección anterior. Para esta colección y, siguiendo nuestro diseño no relacional, hemos implementado una *shared key* con un índice de ordenación por clases, los documentos. De esta manera, cada `chunk` puede almacenar unas clases y, los datos son insertados aleatoriamente en cuanto a su clase, hemos determinado que era idóneo. También hemos tenido en cuenta las *querys* que realizamos, que en su gran mayoría son respecto a la clase de cada documento.

4.3. *Sharding* Outliers

Para la colección que almacena los *outliers* hemos decidido usar una *shared key* respecto al atributo `conf`. Hemos usado este índice ya que todas la querys de la colección dependen de la configuración a la que queremos acceder. Además, el *shard* balanceará de manera correcta la colección, ya que al añadir nuevos datos discriminará por la configuración con la que realicemos la inserción. De esta manera, no creemos que pueda existir *hot spot* o *bottle neck* a lo largo del tiempo.

5. INSERCIÓN DE LOS RESULTADOS

En esta sección realizaremos una serie de **análisis sobre los outliers** que tenemos almacenados en las colecciones. El funcionamiento de estos experimentos se basa en leer los vectores de características, realizar una serie de convoluciones y finalmente registraremos los resultados en dos nuevas colecciones.

- **Experimental**. Se generan registros con un `id` único para clasificar los diferentes experimentos. Cada experimento será registrado con su configuración, método, repetición y parámetros utilizados. Al finalizar, se actualiza añadiendo el nombre del dataset, el número de *outliers*, los verdaderos positivos y el escalar.
- **New_Outliers**. Registra todos los *outliers* que han sido modificados, guardando el *dataset*, la configuración, la iteración del experimento, su `id` real, sus características y la clase a la que pertenecen.

Cabe mencionar también que antes de realizar cualquier experimento con los datasets ya almacenados, hemos generado un bucle que añade un `id` en estas colecciones, puesto que antes solo existia una identificación a través del `objectID`. Para resolverlo, hemos creado en la función `loadVectorData` un id `int` por cada *outlier* de los datasets. Hemos realizado esta modificación debido a que el código proporcionado no estaba implementado de tal manera que reconociese estos `objectID`. A continuación detallamos los aspectos más importantes de nuestra **implementación**.

5.1. insertExperiment

Se genera la colección `Experimental` si aún no existe e insertamos el primer experimento con `id` 0. Si existe, buscamos el último `id` con una query e incrementamos el valor en 1 para no repetir `id` en los experimentos, posteriormente se **registra el experimento** en la colección con sus características.

Listing 4. Formato query experimento

```
"met":str, "conf":str, "numViews":int, "rep":int,
    "experiment":{}
```

5.2. insertOutlierData

Genreamos la colección `New_Outliers` si aún no existe. Iteramos el array `newFeatures` y si existe su `id` en el vector `OutliersGTIdx` se **introduce el *outlier* en la colección**. Para esto, buscamos si el vector de características coincide con algun vector de nuestra colección del *dataset*. En algunos casos, la *label* és `None`, lo que significa que el vector de características no se encuentra en la colección.

Listing 5. Formato query *outliers*

```
"label":str, "rep":int, "conf":str, "dataSet":str
    "id":int, "features":[]
```

5.3. insertResults

Se **actualiza la colección `Experimental`** añadiendo en el documento correspondiente (`_id`) los resultados obtenidos, con la siguiente estructura:

Listing 6. Formato query resultados

```
"fpr":[], "tpr":[], "auc":double, "dataset":str
```

5.4. loadOutliers

Añadimos los *outliers* que hemos insertado en la colección `New_Outliers`. Hacemos una query a través del dataset, la repetición y la configuración, almacenamos todos estos registros en el diccionario de outliers. Si el número de llaves es igual al numero total de *outlires* calculado, significa que los datos se han insertado correctamente y que el experimento puede continuar.

5.5. loadVectorData

Inserta los **vectores característicos** de las colecciones que ya están almacenadas. Realizamos un `find` para obtener todos los datos, seguidamente se ajustan algunas llaves del diccionario para que se correspondan a la estructura de nuestra colección, puesto que la clave `features` en nuestro caso se denomina `vector`.

5.6. loadImageData

Inserta los **descriptores de las imágenes** almacenadas. Realizamos un `find` para obtener todos los datos, seguidamente se ajustan algunas llaves del diccionario para que se correspondan a la estructura de nuestra colección, puesto que la clave `features` en nuestro caso se denomina `descriptor`.

6. VISTAS Y RENDIMIENTO

Las vistas son una parte importante para la resolución del proyecto. Estas resumirán de manera concisa los datos que necesita una persona sobre la colecciones existentes en la base de datos. Se realizaron tres consultas.

6.1. Vista 1

Listing 7. Consulta 1

```
db.createView(
  "OutliersWith 2-0 Conf",
  "New_Outliers", [
    {$match:{conf:"2-0"}}, {$project:{label:1,
        features:1, dataset1}}
  ]
)
```

Esta consulta pide mostrar los *Outliers* que se han realizado con la configuración *2-0*. Como se puede comprobar, proyectamos los datos de interés que son qué outlier es, de qué dataset se ha extraído y cuáles son sus características.

En la colección de Outliers tenemos una gran cantidad de datos, es por eso que hemos realizado una serie de índices para mejorar el tiempo de ejecución de esta query, hemos pensado que la mejor manera es crear un índice ordenado por configuración para que solo tenga que buscar los documentos con esta configuración que ya estarán previamente ordenados. Al realizar dicho índice pasamos de buscar por un `fetch` de casi 50,000 documentos a unos 6500 documentos.

6.2. Vista 2

Listing 8. Consulta 2

```
db.createView(
  "LetterInformation",
  "LetterRecognition",
  [{$project:{label:1,_id:0 }},
    {$group: {_id:null, Classes:{$push: "$label"
        },Total:{$sum:1}}},
    {$unwind: "$Classes"},
    {$group: {_id:"$Classes", "Docs X Classe":{
        $sum:1}, TotalDocs:{$addToSet: "$Total"
        }}},
    {$unwind: "$TotalDocs"}
  ]
)
```

Esta consulta consiste en diferenciar la clases de la colección `letter-recognition`. No obstante, pensamos que la información de interés no era solo cuantas clases hay y cuales son, sino tambié cuántos documentos hay por clase y cuáles son los documentos totales que hay en la colección. Por esta razón decidimos hacer la query un poco más completa de lo que se nos solicitaba. En este caso, hemos realizado un `project` para quedarnos únicamente con los elementos que nos interesan, las clases. Hemos diferenciado las clases del array que se forma y hemos ido jugando con los statges y el comando `$sum` y `$push` para conseguir tenener una vista que muestra lo comentado anteriormente. Respecto al rendimiento de esta query, hemos dividido esta colección y creado los índices pertinentes. Sabiendo que los `aggregate` son complicados para los índices pensamos que era importante quedarnos solo con los atributos específicos para poder mejorar el rendimiento de esta consulta.

6.3. Vista 3

Listing 9. Consulta 2

```
db.createView(
    "Informacio_all_Collections",
    "MirFlickr25K",
    [
    {$facet:{"Iris":[{$lookup:{
        from: "Iris",
        localField: "utility",
        foreignField: "utility",
        as: "Info"
    }},
    {$project:{Info:1, _id:0}}
    {$group:{_id:"$Info"}},
    {$unwind: "$_id"},
    {$group:{_id:null ,Classes:{$push:"$_id.label
        "},TotalDocuments:{$sum:1} }},
    {$unwind:"$Classes"},
    {$project: {Classes:1, TotalDocuments:1}},
    {$group:{_id :"$Classes", "docsXclasse":{$sum
        :1}, TotalDocs:{$addToSet:"
        $TotalDocuments"}}},
    {$unwind: "$TotalDocs"},
    ],


    "BreastCancer":[{$lookup:{
        from: "BreastCancer",
        localField: "utility",
        foreignField: "utility",
        as: "Info"
    }},
    {$project:{Info:1,_id:0}}
```

```
    {$group:{_id:"$Info"}},
    {$unwind: "$_id"},
    {$group:{_id:null ,Classes:{$push:"$_id.label
        "},TotalDocuments:{$sum:1} }},
    {$unwind:"$Classes"},
    {$project: {Classes:1, TotalDocuments:1}},
    {$group:{_id :"$Classes", "docsXclasse":{$sum
        :1}, TotalDocs:{$addToSet:"
        $TotalDocuments"}}},
    {$unwind: "$TotalDocs"}],



    "Ionosphere":[{$lookup:{
        from: "Ionosphere",
        localField: "utility",
        foreignField: "utility",
        as: "Info"
    }},
    {$project: {Info:1,_id:0}}
    {$group:{_id:"$Info"}},
    {$unwind: "$_id"},
    {$group:{_id:null ,Classes:{$push:"$_id.label
        "},TotalDocuments:{$sum:1} }},
    {$unwind:"$Classes"},
    {$project: {Classes:1, TotalDocuments:1}},
    {$group:{_id :"$Classes", "docsXclasse":{$sum
        :1}, TotalDocs:{$addToSet:"
        $TotalDocuments"}}},
    {$unwind: "$TotalDocs"}],



    "MirFlickr25K":[
    {$project:{_id:0, label:1}},
    {$group:{_id:null, label:{$push:"$label"},
        Total:{$sum:1}}},
    {$project:{_id:0, label:1, Total:1}},
    {$unwind:"$label"},
    {$unwind:"$label"},
    {$group:{_id:"$label", DocsxClasse:{$sum:1},
        TotalDocuments:{$addToSet:"$Total"}}},
    {$unwind:"$TotalDocuments"}
    ]}}
    ])
```

Por último, la consulta 3 solicitaba una cosa realmente complicada en varios aspectos: primero, debía ser una única query que mostrara cuantos documentos hay en cada colección de `insertData`, cuantas clases, cuáles y, nosotros hemos añadido, cuántos documentos hay por clase. Para poder Realizar esta query hemos implementado un atributo a cada colección llamado `utility`. Este atributo únicamente tiene la finalidad de poder unir las colecciones con un lookup y poder realizar la query como una sola consulta. Aún así también pensamos que seria útil para poder diferenciar cuales son las colecciones iniciales para los experimentos. Esta query esta formada por varias querys dentro de ella, esto lo hemos podido realizar gracias a la función `$facet`. El acceso a toda la información de todos los datos lo hemos realizado con la función `$lookup`. Respecto al performance de esta query, sabemos que es muy costosa, el timepo de ejecución de la vista es de cerca de 2 minutos. Sabemos que realizar un ínidice sobre esta query no tinene mucho sentido; en los aggregates, los únicos *pipelines* que pueden acceder a ínidices son el `sort`, `project`, `match`, ect... Las funciones `$lookup` y `$facet`, por el contrario, no. Se podria pensar que la query sería más rapida si como elemento del aggregate para hacer el *lookup* se usase la colección que menos datos tiene, esto no se puede hacer porque el array donde se guardan los datos de la colección con la que se realiza el lookup no puede ser mucho más grande que la que estamos utilizando en el aggregate; la principal debe poder incluir a la secundaria, por eso nuestro principal es la colección *MirFlickR25K*. Los `aggregates` en MongoDB son una proyección del primer *stage*, si en este no se usan índices, el resto tampoco lo hará. Es por eso que la performance de esta query no se ha podido mejorar, quizás esto es un problema de fondo respecto a nuestro diseño de la base de datos; no obstante, defendemos la manera en que hemos estructurado nuestra base de datos y pensamos que ha sido la manera más óptima posible.

En esta consulta no se ha incluido la información de la colección *Letter-recognition* porque hay una vista específica para dicha colección.

7. *Scripts* DE VALIDACIÓN

Para realizar este proyecto hemos utilizado una serie de *scripts* para validar las competencias, ejecutar de manera distribuida, facilitar una serie de consultas y comprobar que los datos son correctos. Algunos de los scripts los hemos utilizado en momentos muy concretos de la realización de la práctica y , por ende, no tienen ninguna utilidad más allá de la que se le ha dado en dichos momentos. No obstante, hay otros que son los necesarios para realizar la evaluación del proyecto, como los scripts de `validacio.sh`, con su copia para ejecutar en local, `Consultes.sh`, con su copia para ejecutar en local y `Execution.sh` que es la utlizada para insertar los datos en los diferentes momentos del proyecto. A continuación explicaremos la utilidad y como ejecutarlos.

7.1. Consultes.sh & ConsultesLocal.sh

Ambos *scripts* estan los juegos de querys que hemos realizado para demostrar la correcta inserción de los datos. Solo deben recibir uno de estos parámetros:

- Outliers
- Experiments
- Iris
- LetterRecognition
- BreastCancer
- Ionosphere

7.2. Validació.sh & ValidacióLocal.sh

En estos *scripts* estan las diferentes consultas necesarias para verificar los permisos otorgados a cada uno de los usuarios. Estos scripts pueden recibir hasta 3 parámetros. Si el script recibe un parámetro equivalente al nombre de uno de los usuarios que no son el root, este se concetará a la base de datos con el usuario root y devolverá los privilegios del usuario pasado por parámetro y su rol. El primer parámetro debe ser **GestorDades** o **TEST**.

Si el *script* recibe dos parámetros, el segundo parámetro será cogido como la contraseña de autentificación. En este caso, para demostrar que el los diferentes usuarios **Gestor** y **TEST** solo tienen acceso a la base de datos *Outliers* se enseñará por pantalla la comanda `show dbs`, que muestra las bases de datos visibles para estos usuarios, y se mostrarán las colecciones de esta. Todo esto si la contraseña que se ha pasado como segundo parámetro es correcta. El segundo parámetro debe ser **gestor** si user equivale a *GestorDades* o **test** si es *TEST*. Finalmente, si el script recibe los tres parámetros accederá a una opración Básica de `CRUD`, las cuales el *Gestor* podrá, mientras que el usuario *Test* solo

podrá realizar `Find` e `Insert`. El tercer parámetro tiene que ser uno de los siguientes:

- Find
- Insert
- Remove
- Update

7.3. Execution.sh

Este *script* es el utilizado para insertar los datos de una manera rapida. El insert de los datos lo hicimos desde el *clúster* `main` con el usuario correcto y la inserción de los experimentos lo hemos realizado desde `local`.

7.4. Otros *scripts*

Los demás *scripts* son los que hemos utilizado para facilitar el trabajo de distribuïr la información por los diferentes clusters.

- *Script.sh* Se encarga de copiar los ficheros que le ibamos actualizando a todos los clusters y de apagar y encender los *standAlones* antes de tener el *sharding* y los *replicaSet*.
- *Keys.sh* Solo lo hemos utilizado para copiar el sistema de llaves y evitar introducir las contraseñas cada vez que intentabamos conectarnos a los diferentes mongos.
- Create_User.sh Es el *script* que hemos utilizado para crear el usuario `root` en los diferentes mongos de manera distribuida.
- Replica.sh Este *script* lo usamos para desplegar los *replicaSet* con sus configuraciones correctas en los mongos que no eran de configuración, es decir, los `mongos-1`, `mongos-2`, `mongos-3`, `mongos-4`, `mongos-5` y `main`.

Como se puede observar, hemos realizado numerosos *scripts* para trabajar de una manera más eficaz y facilitar el acceso y la corrección del trabajo.

8. Consideraciones finales

En la evaluación del proyecto no se describre la función `InsertImageOutliers`. Para ejecutar `HOAD` se utiliza el string *COD*, de esta manera se indica en el código base inicial, en esta misma función hemos modificado la procedencia del parámetro `mutual_k`, puesto que tal y como estaba establecido buscaba el valor de esta variable en el documento `metadata.json`, pero este valor se especifica en el documento `configTEST.json`. Se han realizado los experimentos con todos los parametros en los datasets *Iris*, *Ionospehere* y *Letter-recognition*. El número de repeticiones realizadas para el primero es 50, en los restantes, 5; debido a su gran tamaño. Todas las ejecuciones han sido correctas.
Un problema que hemos sufrido al el ejecutar un experimento con el parámetro `-N` sobre cualquier *dataset*, no lo reconoce y ejecuta el *dataset* por defecto; es por eso que cuando hemos realizado los experimentos, hemos modificado el nombre del *dataset* «*Synthetic Data*» por los distintos datasets utilizados.

Referencias

[1] R.A. Fisher, *Iris Data Set*, CA: University of California, School of Information and Computer Science, 1936. [Dataset]. Available: https://archive.ics.uci.edu/ml/datasets/iris.

[2] Space Physics Group, *Ionosphere Data Set*, CA: University of California, School of Information and Computer Science, 1989. [Dataset]. Available: https://archive.ics.uci.edu/ml/datasets/Ionosphere.

[3] Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian, *Breast Cancer Wisconsin (Diagnostic) Data Set*, CA: University of California, School of Information and Computer Science, 1989. [Dataset]. Available: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin.

[4] David J. Slate, Odesta Corporation, *Letter Recognition Data Set*, CA: University of California, School of Information and Computer Science, 1989. [Dataset]. Available: https://archive.ics.uci.edu/ml/datasets/Letter+Recognition.

[5] M. J. Huiskes, *The MIR Flickr Retrieval Evaluation 25K*, Canada: ACM International Conference on Multimedia Information Retrieval, 2008. [Dataset]. Available: http://press.liacs.nl:8080/mirflickr/mirflickr25k.v3/mirflickr25k.zip.

[6] O. Ramos Terrades, *BDnR-1920-grup01*, España: Universidat Autònoma de Barcelona, 2020. [Repository]. Available: https://bitbucket.org/uabgrauinformatica/bdnr-1920-grup01

[7] Dev Ittycheria, Dwight Merriman, MongoDB Inc., *Documents*, EEUU: New York City, New York, 2008. [Online]. Available: https://docs.mongodb.com/manual/core/document

[8] Dev Ittycheria, Dwight Merriman, MongoDB Inc., *Databases and Collections*, EEUU: New York City, New York, 2008. [Online]. Available: https://docs.mongodb.com/manual/core/databases-and-collections

[9] Dev Ittycheria, Dwight Merriman, MongoDB Inc., *Sharding*, EEUU: New York City, New York, 2008. [Online]. Available: https://docs.mongodb.com/manual/sharding/

9. ANEXO

9.1. Estructura de las colecciones

Cuadro 1
Iris, Ionoshpere & Letter-recognition

_id	id	Label	utility	vector
ObjetID	int	str	*Outliers*	array

Cuadro 2
Breast-cancer

_id	id	Label	utility	vector
ObjetID	int	bool	*Outliers*	array

Cuadro 3
MirFlickr25K

_id	id	Descriptors	label	path	utility
ObjetID	int	array	*array*	str	*Outliers*

9.2. Metadatos

Listing 10. Archivo `metadata.json`

```
{
  "iris": {
    "label_pos": 4,
    "k": "3"
  },
  "ionosphere": {
    "label_pos": 34,
    "k": "2"
  },
  "letter-recognition": {
    "label_pos": 0,
    "k": "26"
  },
  "breast-cancer": {
    "id": 0,
    "label_pos": 10,
    "labels": {"2": false, "4": true},
    "k": "2"
  },
  "mirflickr25k": {
    "label_folder" : "anno",
    "images_folder": "images",
    "descriptor_folder": "features",
    "feat_size": "4",
    "type": "f"
  }
}
```

9.3. Distribución del trabajo

Los miembros del proyecto nos hemos dividido el trabajo de tal manera que cada uno realizaba una parte proporcional del mismo, Carlos Boned ha realizado las configuraciones de la base de datos, los *scripts* y las vistas; Miguel Esteban se ha encargado del diseño de las colecciones, la inserción de la información y de la maquetación y parte de la redacción de este documento; por último, Arnau Puche se ha responsabilizado de los experimentos y la gestión de los *outliers*.