Data Engineering
Master in Artificial Intelligence

Anália Lourenço (analia@uvigo.es)
Guillermo de Bernardo (guillermo.debernardo@udc.es)
Pedro Celard (pedro.celard@usc.es)

Dimensional modelling and ETL data processing

1. Software and tools

To complete the assignment, students will have to use

- MongoDB as NoSQL database.
- MongoDB Compass as DB explorer.
- A Python (3.x) installation. You may use libraries and an IDE of your choice.

2. Data sources

The goal of this case is to implement a dimensional model to support the analysis of medical images.

The main data source is a Kaggle dataset, available at the following URL. We will specifically use the **dicom_dir** directory inside the dataset:

https://www.kaggle.com/datasets/kmader/siim-medical-images?select=dicom_dir

Students are strongly encouraged to check the documentation of the dataset available on Kaggle to better understand the structure of the data.

During this lab, we will use the image metadata as the main source of information. In medicine, the DICOM data format is the most used when working with images. Students are encouraged to review the official documentation of the format at https://www.dicomlibrary.com/ and its tags at https://www.dicomlibrary.com/dicom/dicom-tags/

3. Tasks

3.1 Install MongoDB Community Edition

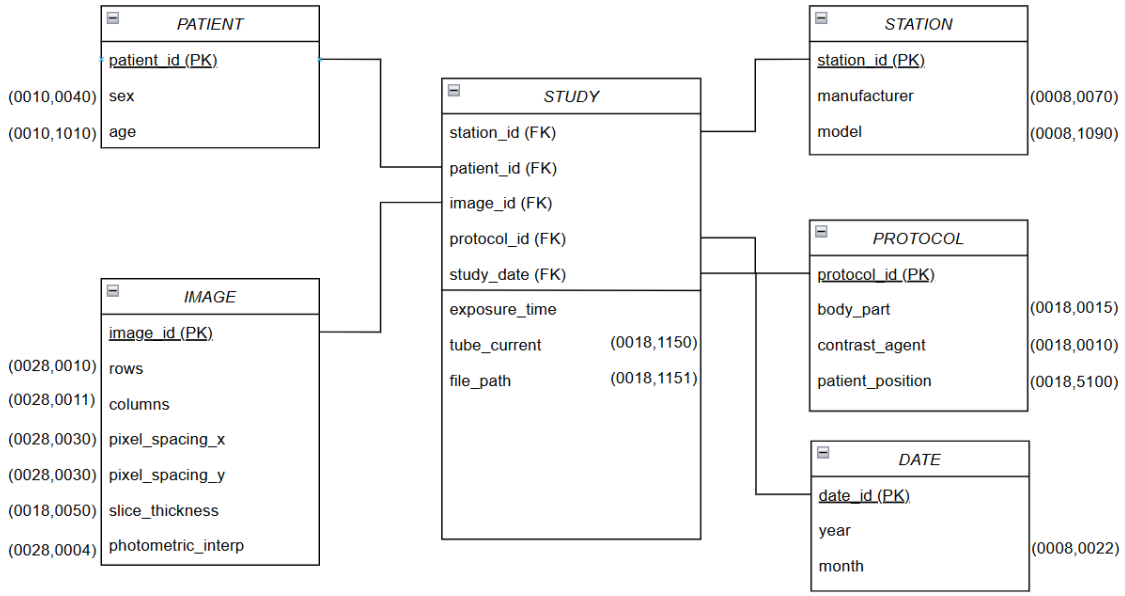
https://www.mongodb.com/try/download/community

3.2 Install MongoDB Compass

https://www.mongodb.com/try/download/compass

3.3 Data analysis and modelling

The objective is to model the medical imaging quality, equipment performance, and protocol consistency across patients, stations, and time. Below you have the star schema model that we will use, along with DICOM codes that will be useful to you later.



3.4 Possible python libraries

Since this is a more guided exercise, the use of the following Python libraries is recommended; however, as in the previous exercises, the use of any other library is also allowed.

- NumPy (**numpy**) – Linear algebra and numerical operations.
- Pandas (**pandas**) – Data processing and CSV file handling.
- Seaborn (**seaborn**) – Data visualization.
- Matplotlib (**matplotlib.pyplot**) – General plotting and image display.
- OS (**os**) – File and directory operations (paths, listing files, etc.).
- Hashlib (**hashlib**) – Secure hash algorithms (e.g., MD5, SHA) for data integrity checks.
- pydicom (**pydicom**) – Reading, writing, and processing DICOM medical imaging files.
- PyMongo (**pymongo**) – Interface to connect and interact with MongoDB databases.
- Scikit-image (**skimage.io**) – Reading images into NumPy arrays.
- Glob (**glob**) – Finding pathnames that match a specified pattern.
- Datetime (**datetime**) – Working with dates and times.
- Pillow (**PIL**) – Image processing and manipulation.

3.5 How to Read DICOM data

DICOM stands for Digital Imaging and Communications in Medicine. It is the standard format used in hospitals for medical images such as CT scans, MRIs, and X-rays. Each file includes an **image** (for example, a scan slice) and **metadata** such as patient age, scan type, and machine details.

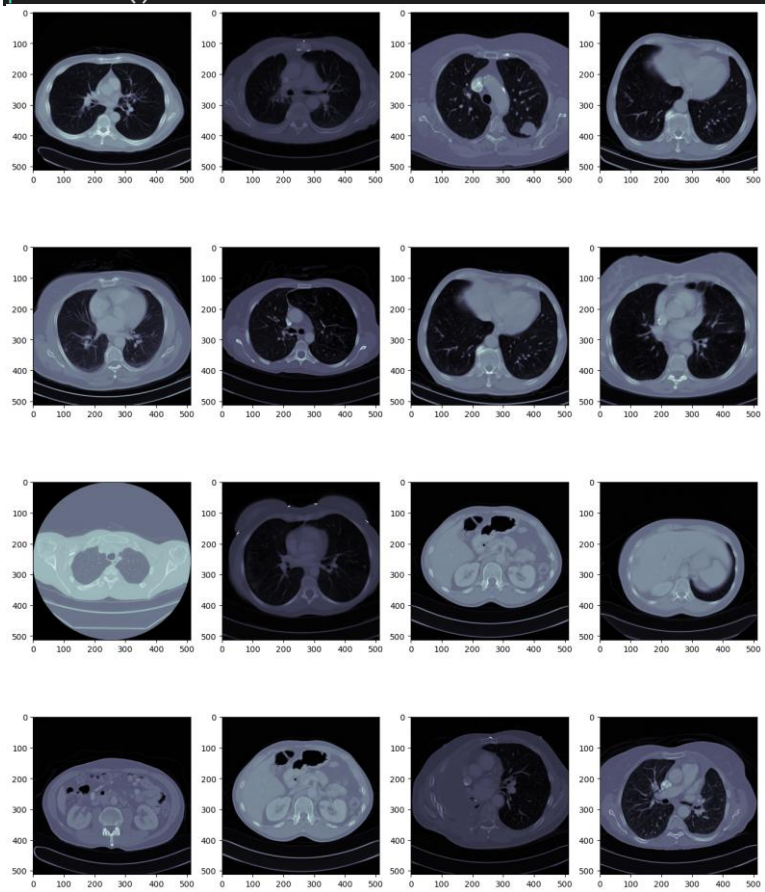
Images in this format are typically 512 × 512 pixels, which is a common size for CT slices. They often appear blueish because of the default grayscale colormap used in medical imaging viewers. Low-intensity pixels appear darker blue or black, representing air or empty space. Medium-intensity pixels appear in gray or blue shades, representing soft tissues such as muscles or organs. High-intensity pixels appear white or very light, representing bone or other dense

structures. This colormap helps highlight anatomical structures and makes interpretation easier for both humans and algorithms.

```
DICOM_FILES_PATH = os.path.join(DATA_PATH, '*.dcm')
dicom_data = pd.DataFrame([{'path': filepath} for file path in glob(DICOM_FILES_PATH)])
dicom_data['file'] = dicom_data['path'].map(os.path.basename)

# Show 16 DICOM images in a 4x4 grid
img_data = list(dicom_data[:16].T.to_dict().values())
f, ax = plt.subplots(4,4, figsize=(16,20))
for i,data_row in enumerate(img_data):
    data_row_img = pydicom.dcmread(data_row['path'])
    ax[i//4, i%4].imshow(data_row_img.pixel_array, cmap=plt.cm.bone)

plt.show()
```



DICOM metadata is the extra information stored alongside each medical image. It describes the patient, scan, and acquisition details, making images meaningful and interoperable between systems.

Each piece of metadata is stored as a (group, element) tag, with a name and a value. Example: (0010, 0010) → "Patient Name". Some of the most relevant metadata elements:

- Patient-related:
    - PatientID → Unique patient identifier
    - PatientAge → Age of the patient

    - PatientSex → Gender
- Study-related:
    - StudyDate → Date of the scan
    - StudyDescription → Description of the study
- Image-related:
    - Modality → Type of imaging (CT, MRI, X-ray, etc.)
    - SliceThickness → Thickness of the scan slice
    - PixelSpacing → Physical spacing of pixels in mm
- Contrast and acquisition:
    - ContrastBolusAgent → Indicates if contrast was used
    - KVP → X-ray tube voltage

DICOM metadata allows software to filter, sort, and analyze images automatically, and ensures that images from different machines or hospitals can be interpreted consistently. We can access this data in the following way.

```
# Show all DICOM metadata of one file
dicom_file_path = list(dicom_data[:1].T.to_dict().values())[0]['path']
dicom_file_metadata = pydicom.dcmread(dicom_file_path)
print(dicom_file_metadata)
```

```
…
(0008,0030) Study Time                  TM: '085723.000000'
(0008,0031) Series Time                 TM: '090344.332000'
(0008,0032) Acquisition Time            TM: '090517.558000'
(0008,0033) Content Time                TM: '090501.593000'
(0008,0040) Data Set Type               US: 0
(0008,0041) Data Set Subtype            LO: 'IMA SPI'
(0008,0050) Accession Number            SH: '2819497684894126'
(0008,0060) Modality                    CS: 'CT'
(0008,0070) Manufacturer                LO: 'SIEMENS'
…
```

3.6 How to work with MongoDB

To connect to a MongoDB database in Python, you can use the MongoClient class from the **pymongo** library. The connection string `'"mongodb://localhost:27017"'` specifies that the MongoDB server is running locally on the default port 27017. Once the client is created, it is good practice to verify the connection by sending a simple command such as `'ping'` to the admin database. If the connection is successful, MongoDB responds to the ping, confirming that the server is reachable. Otherwise, an exception is raised, which can be caught and printed to help identify connection issues such as incorrect server addresses or authentication problems.

```
client = MongoClient("mongodb://localhost:27017")
```

```
# Send a ping to confirm a successful connection
try:
    client.admin.command('ping')
    print("Successfully connected to MongoDB")
except Exception as e:
    print(e)
```

To insert data into a MongoDB collection, you can use either the insert_one() or insert_many() methods provided by the pymongo library, depending on whether you want to add a single document or multiple documents at once. The insert_one() function is used to add a single document (Python dictionary) into a collection. For example:

```
patient = {
    "PatientID": "12345",
    "PatientAge": 61,
    "PatientSex": "M"
}

db = client["database_name"]
collection = db["collection_name"]

result = collection.insert_one(patient)
```

In MongoDB, documents are stored in collections. To find a specific document, you can use the find() and findOne() methods, which returns the first document that matches the query criteria. You can use the id as a unique identifier:

```
db.collectionName.findOne({ _id: ObjectId("68ebc70302e7814710790a72") });
```

You can also combine criteria, for example, to ensure both sex and age match:

```
db.collectionName.find({ sex: "M", age: 60 });
```

The main difference is that find will return all coincidences, while findOne will return the first found.

3.7 Data pipeline

Design and implement the data engineering pipelines that populate the data model:

- A high-level description of the pipelines, explaining the steps to read the source data, clean and transform the data and load the final data into the corresponding schema.
- (Optional) Any description of the source or intermediate data, statistics or any other results that justify your design choices.
- The Python source code that implements the pipeline.
- A clear and concise description of how to run the whole pipeline.

In this activity, you will implement several Python functions that will be used in a DICOM (medical imaging) ETL pipeline. These functions help create surrogate keys, manage dimension records, transform metadata, and normalize image data.

By the end of this exercise, your code should reproduce the following set of functions (shown below), which together form a foundation for transforming and preparing DICOM metadata and images for a data warehouse.

Generate Surrogate Keys

- Create a function surrogate_key(values) that receives a dictionary and returns a unique hash string.
- You can use the hashlib library with the MD5 algorithm.
- Ensure the same combination of values always produces the same key.

Insert or Retrieve Dimension Records

- Define a function get_or_create(collection, values, pk_name) that checks if a record exists in a MongoDB collection.
- If it does not exist, insert a new one using the surrogate key as a primary key.
- Return the surrogate key in both cases.

Format Patient Age

- Write format_age(age_str) to transform a DICOM age string such as '061Y' into an integer (e.g., 61).
- Handle missing or malformed data safely.

Convert DICOM to JPEG

- Implement dicom_to_jpeg(input_path, output_dir, size) and save the images in a new folder. The new paths must be saved in the fact table:
    - Read a DICOM file using pydicom.
    - Normalize pixel values to 0–255.
    - Resize to 256x256 and save the image as a grayscale JPEG.
- Ensure the output directory exists and the image name corresponds to the input file name.

Normalize Pixel Spacing

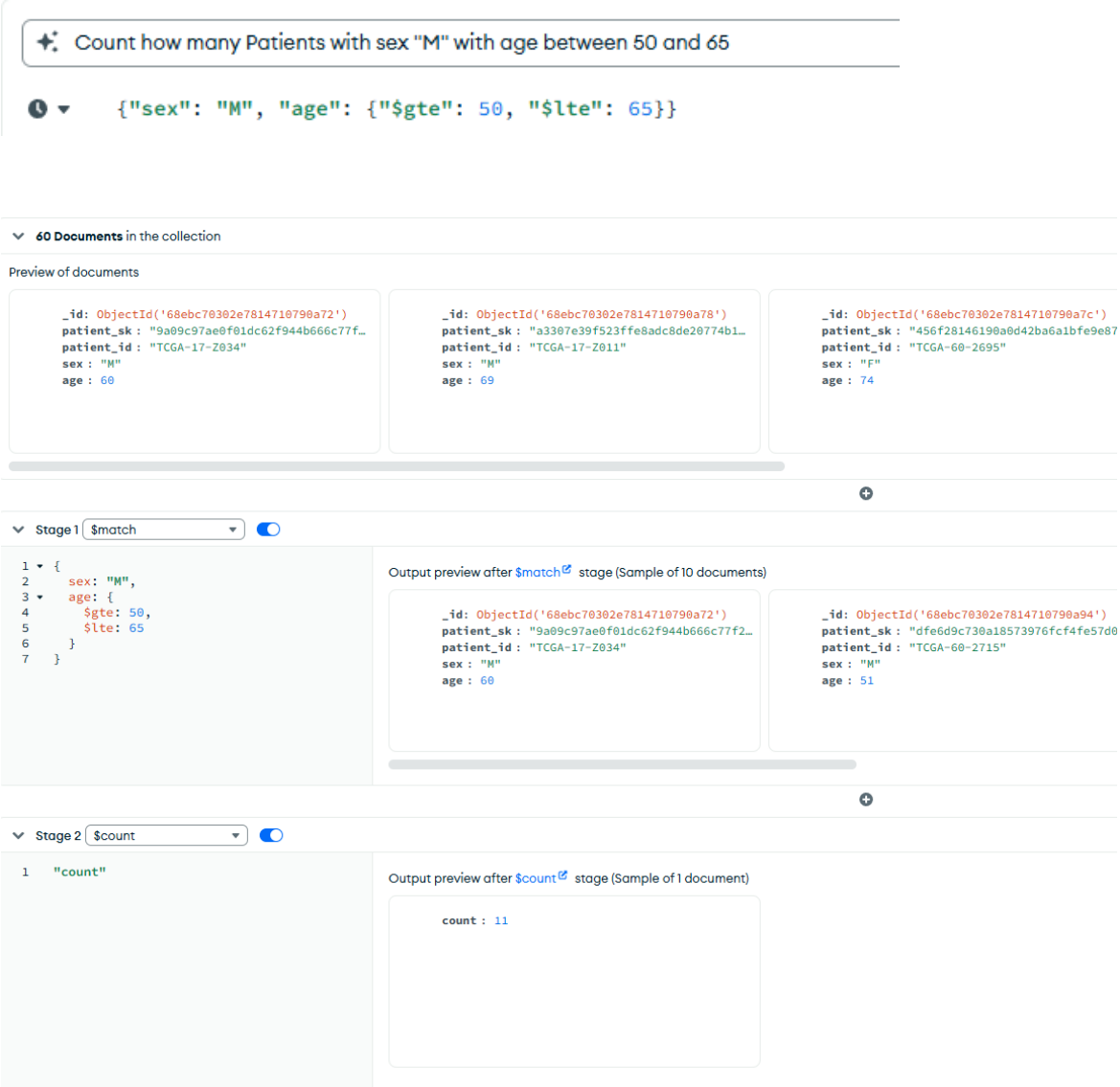
- Create normalize_pixel_spacing(raw_value) to round a numeric pixel spacing value to the nearest bin within a predefined set of values (0.6, 0.65, 0.7, 0.75, 0.8).

Normalize Contrast Agent Field

- Write normalize_contrast_agent(val) to standardize DICOM contrast agent metadata:
    - Replace missing, empty, or single-character values with "No contrast agent".
    - Otherwise, return the cleaned string.

4. Accessing Data

Once the DW is populated with data, it is accessible using MongoDB Compass. We can visualize it using the graphic interface or writing queries. Compass offers an IA tool to write queries, try it with some simple examples.



MongoDB's **aggregation framework** allows you to process data in multiple stages, like SQL queries. Each stage transforms the documents in a collection and passes the results to the next stage. Each stage is an object like $match, $group, $lookup, $unwind, $project, etc.

The basic structure is:

```
db.collection.aggregate([
        { stage1 },
        { stage2 },
         ...
]);
```

The basic stages are:

a) **$match**: Filters documents based on a condition (like SQL WHERE). Only documents that meet the criteria are passed to the next stage.

Example: select patients older than 50

```
{ $match: { age: { $gt: 50 } } }
```

b) **$lookup:** Performs a join with another collection. Combines documents from the “local” collection with matching documents in a “foreign” collection.

```
{
  $lookup: {
    from: "foreignCollection",     // the collection to join
    localField: "localKey",        // field in current collection
    foreignField: "foreignKey",    // field in foreign collection
    as: "outputArray"              // name of the resulting array
  }
}
```

c) **$unwind**: Converts an array into individual documents. Useful after $lookup to work with each joined document separately. Example: after joining dim_protocol, protocol is an array. $unwind flattens it. If the array had 2 elements, one input document becomes 2 documents, each with one element of the array.

```
{ $unwind: "$protocol" }
```

d) **$group**: Aggregates documents based on a key (like SQL GROUP BY). You can compute sums, averages, counts, min/max, etc.

```
{
  $group: {
    _id: "$fieldToGroupBy",            // the key to group documents
    count: { $sum: 1 },                // example: count documents
    avgValue: { $avg: "$someField" }  // example: average
  }
}
```

e) **$project**: Shapes the final output. Can include or exclude fields, create new fields, or rename them. Example: only show patient_id and age:

```
{
  $project: {
    _id: 0,         // exclude MongoDB internal _id
    patient_id: 1,
    age: 1
  }
}
```

f) **$sort:** Sorts documents by a field (ascending or descending).

```
{ $sort: { age: -1 } }
```

As an example, we have the following query, which works as follows:

1. **Joins** fact_table with dim_protocol to get protocol details for each scan.
2. **Flattens** the protocol array so each protocol is treated as a separate row.
3. **Groups** by body_part and **counts** how many scans exist per body part.
4. **Sorts** the results in descending order by count.

Result: You get a **ranked list of body parts** by the number of scans in our dataset.

```
db.fact_table.aggregate([
  { $lookup: {
      from: "dim_protocol",
      localField: "protocol_sk",
      foreignField: "protocol_sk",
      as: "protocol"
  }},
  { $unwind: "$protocol" },
  { $group: {
      _id: "$protocol.body_part",
      count: { $sum: 1 }
  }},
  { $sort: { count: -1 } }
]);
```

5. Submission and evaluation

This lab practice should be conducted in **groups of 2-3 people**.

You are expected to submit the following artefacts for evaluation:

- A PDF report containing:
  - Explanations and diagrams required in each of the tasks.
  - Instructions on how to run the code.
  - Explanation and visual example of each collection created in MongoDB.
  - **A set of questions based on the model and their corresponding query and its explanation to answer the questions.**
- A .zip file with the complete source code and additional files required to run the whole data pipeline. This should include:
  - (Mandatory) Python files that implement the data transformations and load the data into MongoDB.
  - (Optional) Any additional files that may be useful to set up and run the code (e.g requirements.txt, additional README files)

Keep in mind that the source code you provide must be execution-ready on the professor's machine. This means that:

- A small set of configuration parameters **may** be included (preferably in environment or configuration files). This includes the database connection parameters, and possibly the (relative path) location of the input files on the local machine. These configuration steps **must** be clearly documented in your report.
- The source code **must not** contain any absolute local file paths or other elements that prevent it from running on a different machine.
- In summary: make sure that your code can be executed on any other machine following your submitted instructions.

The deadline for submission is **November 2th, 2025, at 23:59.**

You should submit the contents using the available submission task on the online campus of your university.