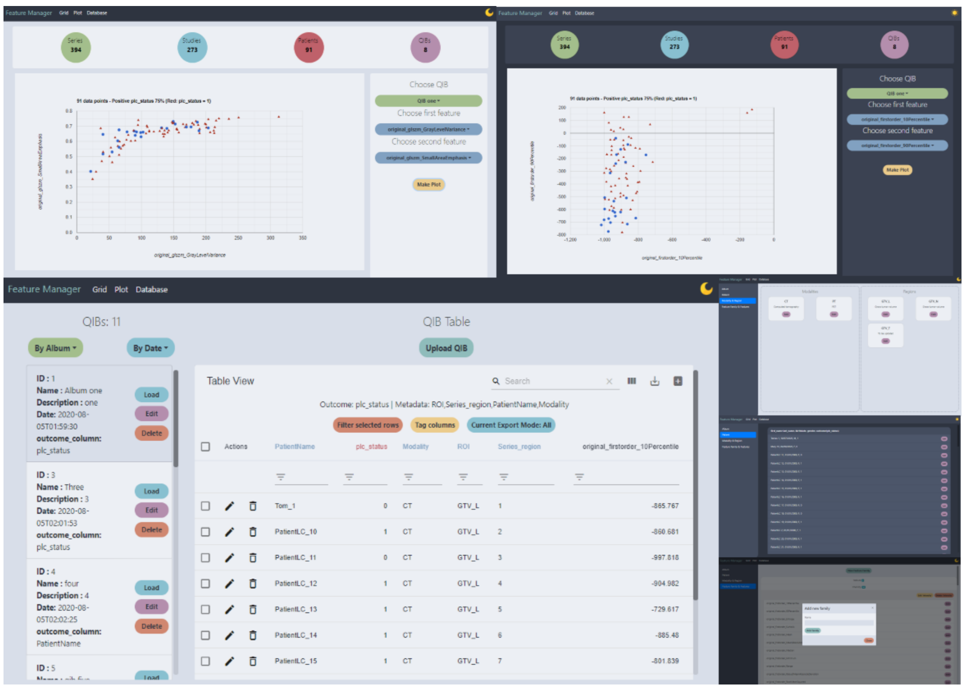
WEB-BASED PLATFORM FOR MANAGING IMAGE BIOMARKERS

August 2020



ID : 1
Name : Album one
Description : one
Date: 2020-08-05T01:59:30
outcome_column: plc_status

ID : 3
Name : Three
Description : 3
Date: 2020-08-05T02:01:53
outcome_column: plc_status

ID : 4
Name : four
Description : 4
Date: 2020-08-05T02:02:25
outcome_column: PatientName

ID : 5

Table View

Outcome: plc_status | Metadata: ROI,Series_region,PatientName,Modality

Actions	PatientName	plc_status	Modality	ROI	Series_region	original_firstorder_10Percentile
	Tom_1	0	CT	GTV_L	1	-865.767
	PatientLC_10	1	CT	GTV_L	2	-860.681
	PatientLC_11	0	CT	GTV_L	3	-997.818
	PatientLC_12	1	CT	GTV_L	4	-904.982
	PatientLC_13	1	CT	GTV_L	5	-729.617
	PatientLC_14	1	CT	GTV_L	6	-885.48
	PatientLC_15	1	CT	GTV_L	7	-801.839

Student : Nghi Tran

Professor : Adrien Depeursinge

ABSTRACT

The aim of this paper is to describe the process of implementing a web-based platform to manage image biomarkers, with a focus on managing and sorting datasets for machine learning.

The paper will first discuss about the emerging field of radiomics, the need for a comprehensive way to manage large datasets of image biomarkers, and the current solutions in the field. This part will be the foundation for the development of this platform. During the first part, the paper will also explain this project's approach to the problem of biomarkers management in contrast to existing solutions, along with a justification of technologies used.

This thesis project aim to develop a web platform to manage sets of quantitative image biomarkers. While solutions to extract these biomarkers have been developed, efficiently storing and managing the extracted data is a challenge.

The project will implement a full stack solution from database, to server to front end application. It transforms extracted image biomarker sets into an interactive web interface for data viewing, exporting, and management.

The deliverables of this project are a relational data model, a back end application with ETL and API functionalities, and a web application.

The result of this project demonstrates that managing quantitative image biomarkers using relational entity model is feasible, but there is still room for improvement.

Keywords: *Healthcare, Radiomics, Data management, Data modelling, ETL, Web*

FOREWORD

This thesis was completed as the final project of the bachelor's degree program in Business Information Technology at HES-SO Valais-Wallis. The thesis was given by professor Adrien Depeursinge and guided by him along with Roger Schaer and Orfeas Aidonopoulos.

The context of this topic is from Swiss Personalized Health Network (SPHN)'s project "IMAGINE", a project with the goal of developing infrastructure for national image-based personalized medicine.

Development of the thesis project spanned from February 2020 to July 2020 in Sierre, Switzerland and later in Helsinki, Finland. The project aims to deliver a working demonstration of the goals provided.

I would like to thank Adrien Depeursinge, Roger Schaer, and Orfeas Aidonopoulos for their guidance in the making of this thesis, and Catherine Tacchini and Isabelle Fournier for helping me coordinate remote work when I moved back to Helsinki.

TABLE OF CONTENTS

TABLE OF FIGURES

ABBREVIATIONS

ACID - Atomicity, Consistency, Isolation, Durability

AGPL - Affero General Public License

API - Application programming interface

CRUD - Create, read, update and delete

CT - Computed tomography

DB - Database

DICOM - Digital Imaging and Communications in Medicine

GTV - Gross tumour volume

IOP - Input/output operations per second

JSON - JavaScript Object Notation

JSX - JavaScriptXML

PT - Short for PET, positron emission tomography

QIB - Quantitative Image Biomarkers

RDS - Relational Database Service

REST - Representational State Transfer

ROI - Region of Interest

RSNA - Radiological Society of North America

SPHN - Swiss Personalized Health Network

SQL - Structured Query Language

1. INTRODUCTION AND STATE OF THE ART

1.1 The field of radiomics

Advances in computing have led to changes in the field of medical imaging, particularly in cancer care. Once largely a qualitative diagnostic tool (Sara Ranjbar, 2017, p. 223), that is, a tool to provide on-hand information to aid in decision-making, a new branch of imaging research has emerged thanks to leaps in computational performance ,which has enabled the large-scale extraction and management of medical images.

Radiomics, as it is called, is the field of medical research where high-throughput data is extracted from large numbers of imaging data that can come from multiple sources and patient profiles. The advantage of radiomics lies in number, whereby levying the sheer amount of data available (patients go through imaging multiple times during their treatment), researchers can glean quantitative imaging features from the images using computer image detection technologies. Radiomics finds its role in cancer treatment as a non-invasive enhancement, but not replacement, to more invasive traditional procedures during the process of diagnosis and assessment.

1.1.1 Biomarkers and features

Biomarkers are indicators of normal or abnormal biologic processes (Sara Ranjbar, 2017). An example of a well-known biomarker is high body temperature as indication of fever. In the context of cancer treatment, the main source of biomarkers come from biopsy samples. Features are a category of measurement in the process of gathering biomarker information. Continuing with the above example, body temperature in Celsius is a feature.

Data from radiomics research not only can act as potential biomarkers, but due to its quantitative approach, can also be used to assess feature robustness, determine the error margin of measuring equipment on a large scale, and perform other meta-purposes (Sara Ranjbar, 2017, p. 229).

Types of features in the scope of this paper: Intensity and Texture features

- Texture features: features that depicts the textural characteristics of tumours.
- Intensity features: features that depicts the intensity of pixels in specific regions

1.2 State of the art

1.2.1 General workflow of radiomics research

The general workflow of radiomics is:

- Image acquisition
- Identification and segmentation of regions of interest
- Quantitative image feature extraction
- Data mining and informatics analysis.

There exist several solutions already on the market that comprehensively covers most of the steps in this workflow. Section 1.2.2 will discuss these solutions.

1.2.2 Existing radiomics solutions

1.2.2.1 I2b2

I2b2[1], short for Informatics for Integrating Biology & the Bedside, is an open source medical data warehouse. Developed by Partners Healthcare and Harvard University and now hosted by the TranSMART Foundation, i2b2 focuses on analytics of biological data, i.e. biomarkers in genomics and clinical data. I2b2 is a full package solution with a self-deployable Java server and a PHP web client.

I2b2 uses a star schema model, a relational database model with a central fact table pointing to multiple dimension tables.

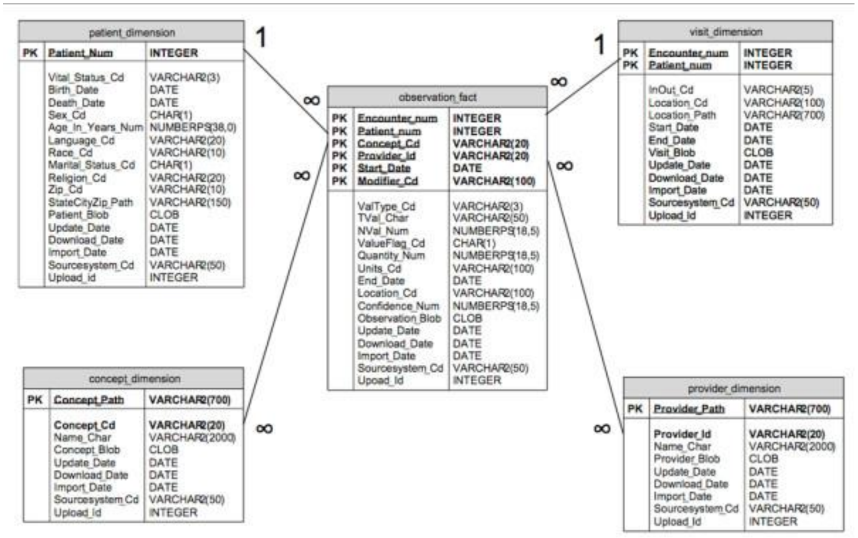

patient_dimension
PK Patient_Num INTEGER
Vital_Status_Cd, Birth_Date, Death_Date, Sex_Cd, Age_In_Years_Num, Language_Cd, Race_Cd, Marital_Status_Cd, Religion_Cd, Zip_Cd, StateCityZip_Path, Patient_Blob, Update_Date, Download_Date, Import_Date, Sourcesystem_Cd, Upload_id
observation_fact
PK Encounter_num, PK Patient_num, PK Concept_Cd, PK Provider_Id, PK Start_Date, PK Modifier_Cd
ValType_Cd, TVal_Char, NVal_Num, ValueFlag_Cd, Quantity_Num, Units_Cd, End_Date, Location_Cd, Confidence_Num, Observation_Blob, Update_Date, Download_Date, Import_Date, Sourcesystem_Cd, Upoad_Id
visit_dimension
PK Encounter_num, PK Patient_num
InOut_Cd, Location_Cd, Location_Path, Start_Date, End_Date, Visit_Blob, Update_Date, Download_Date, Import_Date, Sourcesystem_Cd, Upload_Id
concept_dimension
PK Concept_Path VARCHAR2(700)
Concept_Cd, Name_Char, Concept_Blob, Update_Date, Download_Date, Import_Date, Sourcesystem_Cd, Upload_Id
provider_dimension
PK Provider_Path VARCHAR2(700)
Provider_Id, Name_Char, Provider_Blob, Update_Date, Download_Date, Import_Date, Sourcesystem_Cd, Upload_Id
1
∞

Figure 1-1 Example of a star schema model

The "i2b2 Software" package is made up of 3 components that can be downloaded from the i2b2 website:

- i2b2 Workbench (client)
- i2b2 VMWare (virtual machine Image of a complete i2b2 Server installed on CentOS)
- i2b2 Source (collection of the i2b2 source code for the i2b2 clients and server)

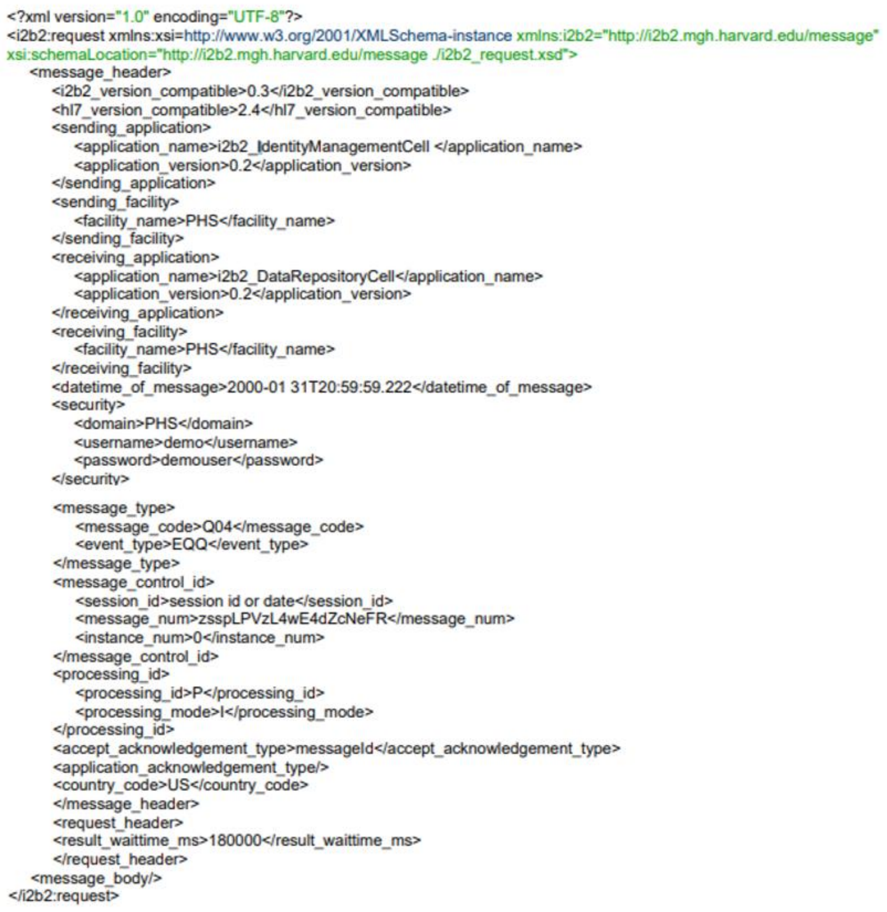
The web client can run on most modern browsers (Chrome, Firefox, Safari) and Microsoft Internet Explorer. To deploy the server, the following software are needed:

- Java (7.0)
- JBoss (7.1.1) for App Server management,
- Apache Ant (1.8.2), Java library and command-line tool used by the i2b2 to drive processes defined in the i2b2 build files.
- Apache Axis2 (1.6.2), Web Services / SOAP / WSDL engine used by the i2b2 web services

A configured database that is either an Oracle, PostgreSQL, or SQL Server database set up with a star schema entity model is also needed.

[1] I2b2's main page can be found here: https://www.i2b2.org

I2b2 treats all data as multiple Cell units, with different categories: Ontology management cells, workplace cells, file repository cells, etc. Communication between cells is carried out in xml format.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<i2b2:request xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance xmlns:i2b2="http://i2b2.mgh.harvard.edu/message"
xsi:schemaLocation="http://i2b2.mgh.harvard.edu/message ./i2b2_request.xsd">
    <message_header>
        <i2b2_version_compatible>0.3</i2b2_version_compatible>
        <hl7_version_compatible>2.4</hl7_version_compatible>
        <sending_application>
            <application_name>i2b2_IdentityManagementCell </application_name>
            <application_version>0.2</application_version>
        </sending_application>
        <sending_facility>
            <facility_name>PHS</facility_name>
        </sending_facility>
        <receiving_application>
            <application_name>i2b2_DataRepositoryCell</application_name>
            <application_version>0.2</application_version>
        </receiving_application>
        <receiving_facility>
            <facility_name>PHS</facility_name>
        </receiving_facility>
        <datetime_of_message>2000-01 31T20:59:59.222</datetime_of_message>
        <security>
            <domain>PHS</domain>
            <username>demo</username>
            <password>demouser</password>
        </security>

        <message_type>
            <message_code>Q04</message_code>
            <event_type>EQQ</event_type>
        </message_type>
        <message_control_id>
            <session_id>session id or date</session_id>
            <message_num>zsspLPVzL4wE4dZcNeFR</message_num>
            <instance_num>0</instance_num>
        </message_control_id>
        <processing_id>
            <processing_id>P</processing_id>
            <processing_mode>I</processing_mode>
        </processing_id>
        <accept_acknowledgement_type>messageId</accept_acknowledgement_type>
        <application_acknowledgement_type/>
        <country_code>US</country_code>
        </message_header>
        <request_header>
        <result_waittime_ms>180000</result_waittime_ms>
        </request_header>
    <message_body/>
</i2b2:request>
```

Figure 1-2 A sample request

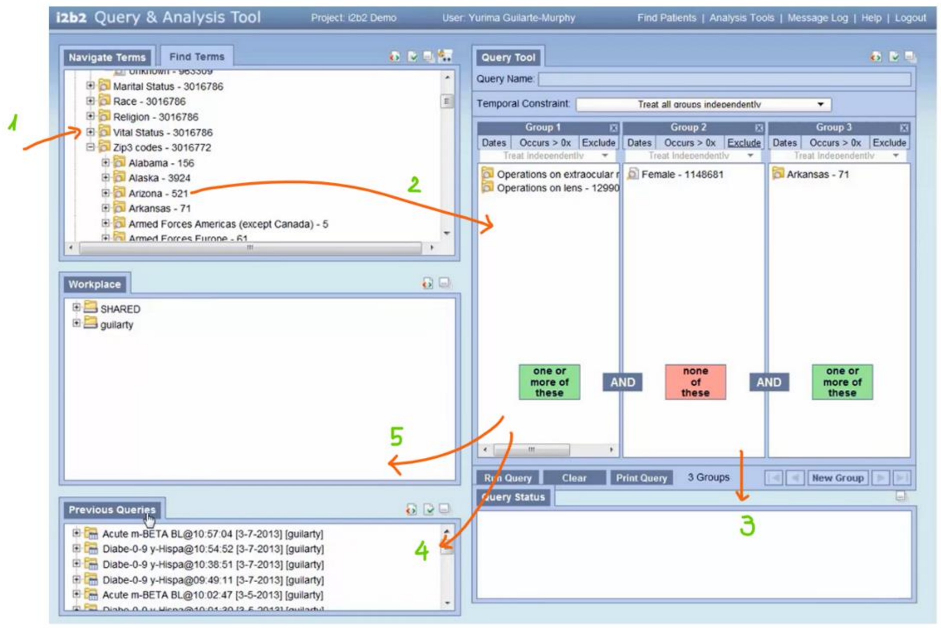


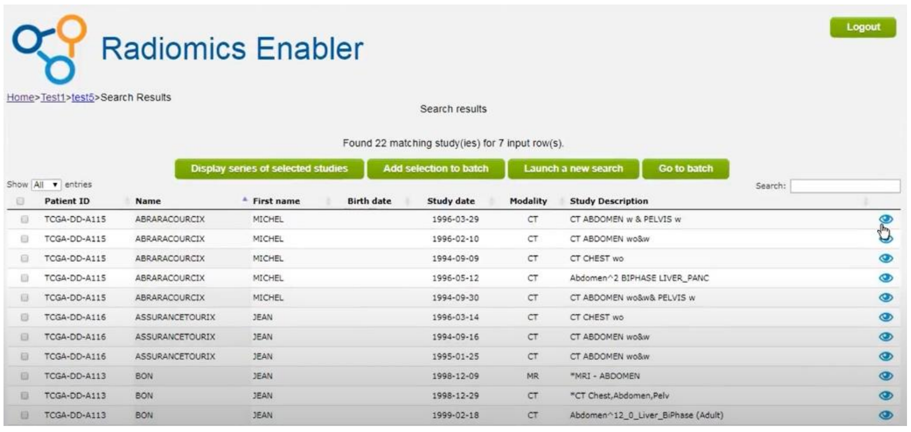
Figure 1-3 i2b2 web client interface

I2b2 has a robust web client system. As indicated in Figure 4, some functionalities of the web client are:

1. Query Term: each correspond to the ontologies cell (data model) in data warehouse
2. Drag-n-Drop terms to Query tool to build query
3. Query result
4. Previous query can be accessed
5. Queries can be stored/ shared with other users

1.2.2.2 Radiomics Enabler

Radiomics Enabler[2] is an opensource (AGPL licensed) web server that can connect to clinical data warehouses (such as i2b2) using the DICOM protocol and combine with RSNA's Clinical Trials Processor to perform ETL for large scale projects. The software is developed by Medexprim, a French start-up founded in 2015. Together with i2b2, it can form a workflow for radiomics research.



Search results

Found 22 matching study(ies) for 7 input row(s).

Patient ID	Name	First name	Birth date	Study date	Modality	Study Description
TCGA-DD-A115	ABRARACOURCIX	MICHEL		1996-03-29	CT	CT ABDOMEN w & PELVIS w
TCGA-DD-A115	ABRARACOURCIX	MICHEL		1996-02-10	CT	CT ABDOMEN wo&w
TCGA-DD-A115	ABRARACOURCIX	MICHEL		1994-09-09	CT	CT CHEST wo
TCGA-DD-A115	ABRARACOURCIX	MICHEL		1996-05-12	CT	Abdomen^2 BIPHASE LIVER_PANC
TCGA-DD-A115	ABRARACOURCIX	MICHEL		1994-09-30	CT	CT ABDOMEN wo&w& PELVIS w
TCGA-DD-A116	ASSURANCETOURIX	JEAN		1996-03-14	CT	CT CHEST wo
TCGA-DD-A116	ASSURANCETOURIX	JEAN		1994-09-16	CT	CT ABDOMEN wo&w
TCGA-DD-A116	ASSURANCETOURIX	JEAN		1995-01-25	CT	CT ABDOMEN wo&w
TCGA-DD-A113	BON	JEAN		1998-12-09	MR	*MRI - ABDOMEN
TCGA-DD-A113	BON	JEAN		1998-12-29	CT	*CT Chest,Abdomen,Pelv
TCGA-DD-A113	BON	JEAN		1999-02-18	CT	Abdomen^12_0_Liver_BiPhase (Adult)

Figure 1-4. Radiomics Enabler interface

1.2.3 Breaking down the workflow

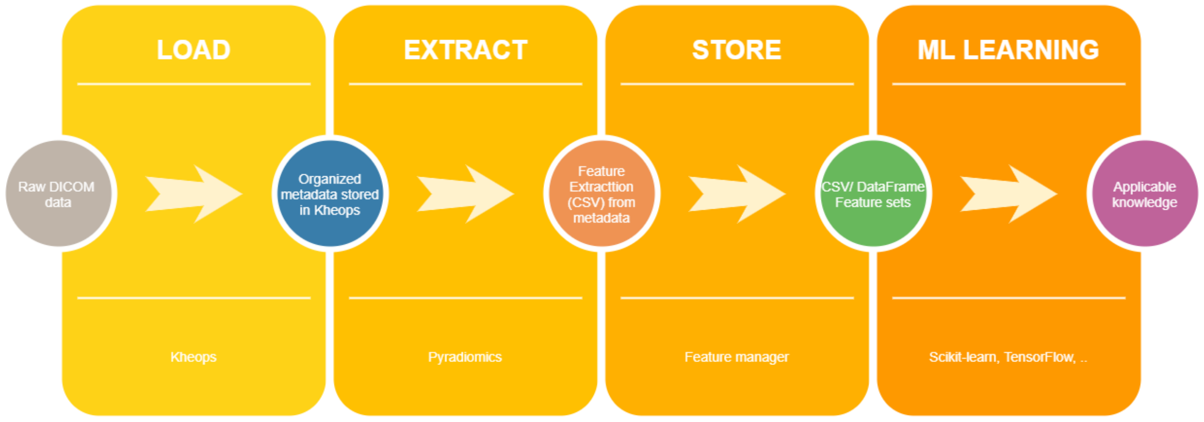

LOAD
EXTRACT
STORE
ML LEARNING
Raw DICOM data
Organized metadata stored in Kheops
Feature Extracttion (CSV) from metadata
CSV/ DataFrame Feature sets
Applicable knowledge
Kheops
Pyradiomics
Feature manager
Scikit-learn, TensorFlow, ..

Figure 1-5 Radiomics workflow by stage and technologies

In order to implement the workflow of radiomics part by part, the following steps are needed:

- Loading: raw DICOM image file input is loaded into a database along with relevant metadata
- Extraction: feature values are extracted from those images using their metadata.
- Storage: Extracted feature sets are stored, to be used for machine learning

[2] Medexprim's product page can be found here: https://www.medexprim.com/radiomics-enabler/

- Machine learning: using machine learning to create models from feature sets.

Section 1.2.4 will discuss the existing solutions on the market that handles each of these individual steps.

1.2.4 Existing specialized products/libraries

1.2.4.1 Kheops

Kheops[3] is an open-source solution for storing and viewing DICOM compliant medical images, developed at Campus Biotech in Geneva, Switzerland.

It is a well-equipped system for managing medical imaging studies, albums, and users, but it is not focused on managing extracted feature sets, which leads to the issue this project aims to address.

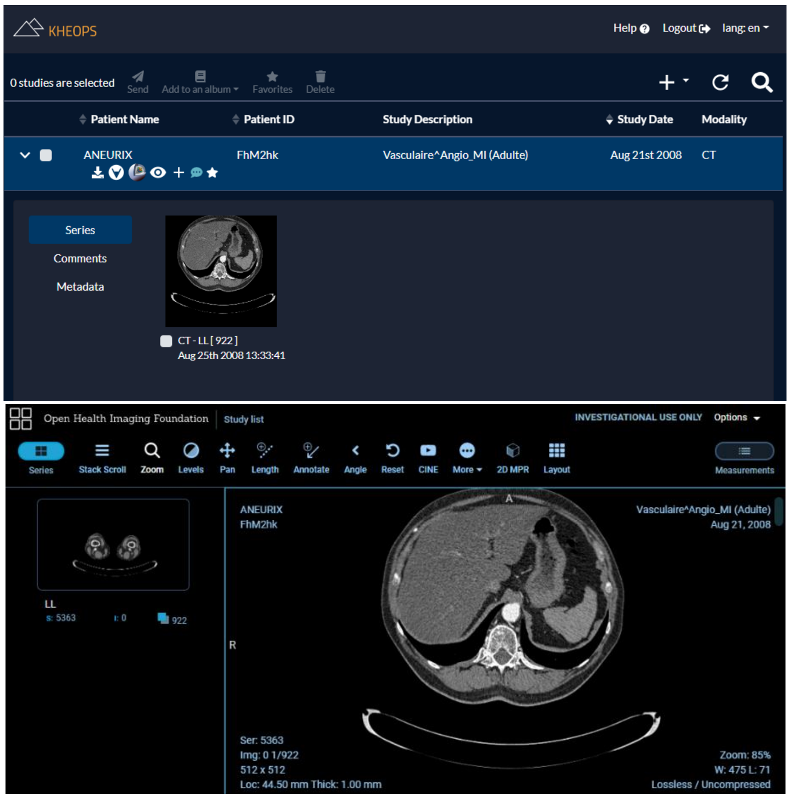


Figure 1-6 Kheops web UI and image viewer

1.2.4.2 Pyradiomics

Radiomics.io is a Boston-based online platform aimed at developing a open-source, standardized benchmark for radiomics projects and a community resource for researchers. One of its most popular project is pyradiomics[4], an (also open source) Python package for radiomics features extraction.

In the current context, pyradiomics is used to extract feature sets using metadata from Kheops and actual DICOM images. The resulting feature sets are exported in CSV format.

[3] Kheops's main page can be found here: https://kheops.online
[4] Pyradiomics's main page can be found here: https://www.radiomics.io/pyradiomics.html

1.2.4.3 Scikit-learn

Scikit-learn[5] is also an open source Python machine learning package for predictive data analysis. It is built on NumPy, SciPy, and matplotlib.

In the current context, training sets are loaded into scikit-learn in CSV formats.

1.3 Role and aim of this project

Currently, extracted feature sets are in CSV format and locally stored until they are needed for machine learning. There is no solution to efficiently store and retrieve these files. To get feature sets that are extracted from a particular album, users would have to manually find that album's metadata in Kheops, then check that with metadata values in the CSV files. To edit a patient's information in the feature set, or a modality name, users would have to manually change them in each CSV file.

This project aims to bridge that gap and create **a feature manager** to:

- Store extracted feature sets
- Provide an interface to query and interactively compile feature sets in real time.
- Edit metadata of feature sets without compromising data integrity
- Implement simple visualization of data

This will help users efficiently view, store, edit, query and compile feature sets for machine learning.

[5] Scikit-learn's main page can be found here: https://scikit-learn.org/stable/

2. METHODOLOGY

2.1 Designing the feature manager

2.1.1 Introduction to terminologies used

This section explains the terminologies used throughout this application. These terms come from various radiomics concepts.

- **QIB:** abbreviation for Quantitative Image Biomarkers. In this model, a QIB represent one feature set (a CSV feature extraction file)
- **Feature:** categories of value measurement/classification. Further explanation can be found in section 1.1.2.
- **Family:** categories of a feature.
- **Modality:** medical imaging procedure used to obtain a biomedical image set
- **ROI:** region of interest.
- **Series:** a series is a set of related DICOM images. Each series is defined by its modality and may contain one or more regions of interest.
- **Study:** a study is a set of series from the same patient.
- **Album:** an album is a collection of studies.
- **Patient:** each patient can be referenced by many studies, but in the scope of this project, each will only have one Outcome.
- **Outcome:** each outcome refers to one patient. In the scope of this project, only plc_status (Pulmonary Lymphangitic Carcinomatosis status: status of tumours in the lung’s lymphatic vessels, whether if they are spreading or not) (Naim Qaqish, n.d.) is used as a binary outcome variable.

2.1.2 Use cases

From the frontend application, users will be able to interact with data in the database in the following use cases:

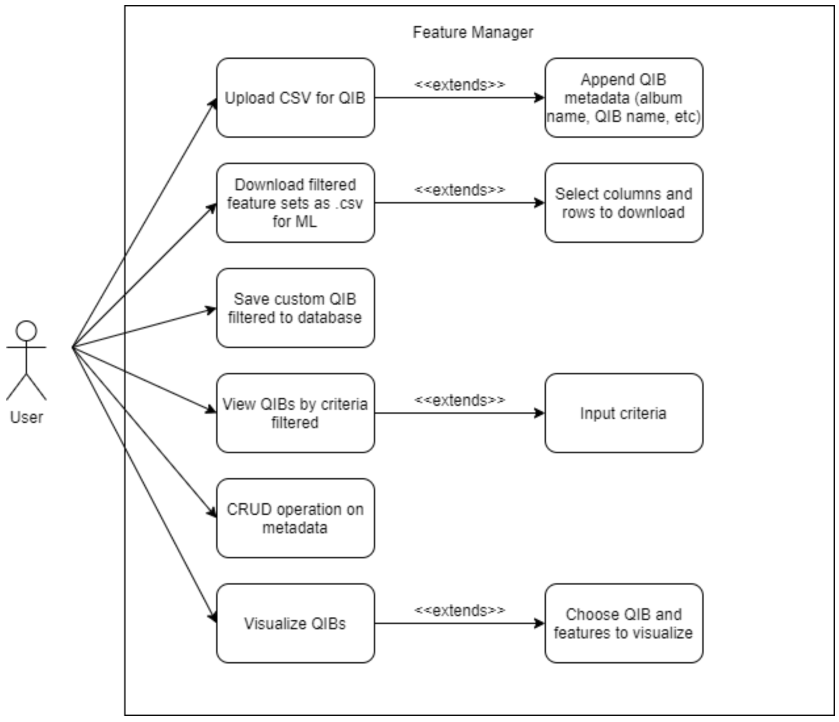

Feature Manager
Upload CSV for QIB
<<extends>>
Append QIB metadata (album name, QIB name, etc)
Download filtered feature sets as .csv for ML
<<extends>>
Select columns and rows to download
Save custom QIB filtered to database
View QIBs by criteria filtered
<<extends>>
Input criteria
CRUD operation on metadata
Visualize QIBs
<<extends>>
Choose QIB and features to visualize
User

Figure 2-1 Use case diagram

- Upload CSV: Users can upload the feature sets extracted from Kheops into the database. Requires users to input the name of the album the set is extracted from, as well as provide a name and description for the uploaded file.
- View the feature sets by criteria: Users can filter the sets by album and by date, as well as manipulate the actual feature set table to filter data by each column (Region of interest, Modality, etc.).
- Download feature sets: Users can download feature sets (filtered/unfiltered) as CSV.
- Saved custom filtered feature set: Users can save their selection for future viewing by uploading the filtered CSV. feature set back to the application.
- CRUD operation on metadata: Users can perform some limited operations to edit albums, feature family's name and description, etc. As a note, CRUD operations on metadata are limited to some entities to preserve the integrity of data.
- Visualize feature sets: Users can visualize a chosen feature set by comparing any two features in that set. The values will be mapped onto a bivariate scatterplot.

2.1.3 Solution overview

It was decided from the start of the project that a relational database model would be used to store the CSV files. By mapping metadata in the extracted features set into relation entities, this will allow for querying for data by entities, instead of manually sorting the CSV files.

The feature manager consists of 3 components:

1. Database: stores data from the CSV files
2. Backend application + API: handles ETL process and provides a RESTful API
3. Frontend application: displays data using the API above, provides an interface for users to upload CSV and filter/download feature sets

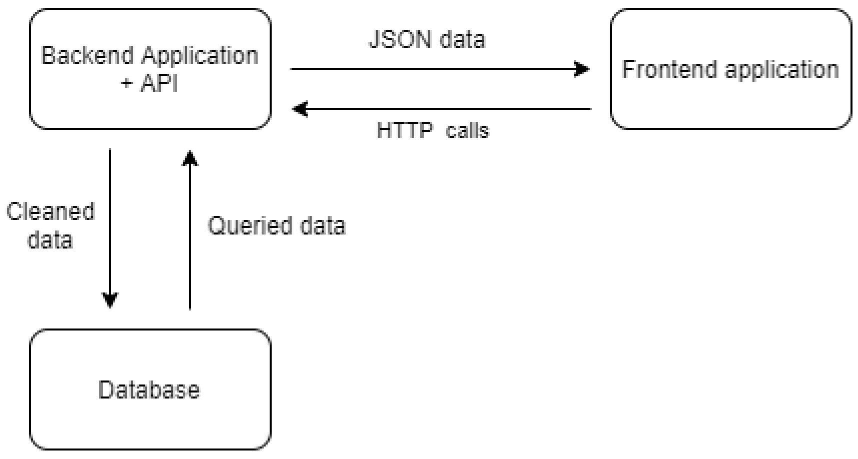


Figure 2-2 General architecture of the feature manager

2.2 Technologies chosen

This section explains the technologies used to implement the feature manager, and makes a comparison with i2b2 (discussed in section 1.2.2.1), since i2b2 also implements some similar features with this project.

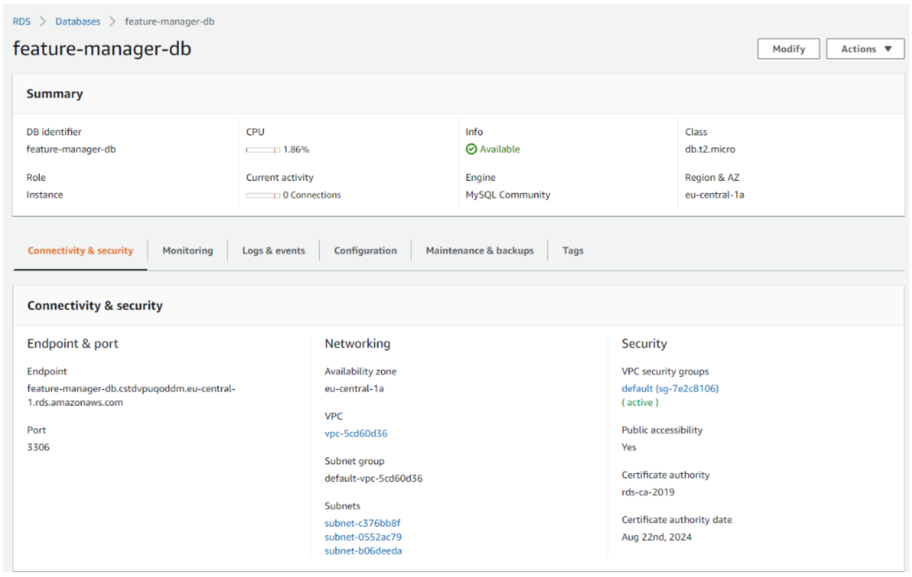
2.2.1 Database

2.2.1.1 MySQL

MySQL[6] is a popular relational database management system. The reasons for choosing MySQL for the database in this project are:

- Open source: it is open source and mature. MySQL is extensively documented and well supported by many libraries.
- ACID: MySQL transactions are ACID compliant (atomicity, consistency, isolation, durability)
- Server-based: MySQL is multithreaded and can handle requests from multiple processes, as opposed to the serverless SQLite that does not support multiple clients.
- Scalability: MySQL databases can be relatively easy to scale up and migrate. Over the course of this project, data has been hosted both online and locally (an Amazon AWS RDS db.micro.t2 instance in Frankfurt, and a local MySQL server). There is no noticeable difference switching between the two except for speed (since the online instance is a Free tier instance, speed is limited).

[6] MySQL’s main page can be found here: https://www.mysql.com



feature-manager-db

Summary

DB identifier: feature-manager-db
CPU: 1.86%
Info: Available
Class: db.t2.micro
Role: Instance
Current activity: 0 Connections
Engine: MySQL Community
Region & AZ: eu-central-1a

Connectivity & security

Endpoint & port
Endpoint: feature-manager-db.cstdvpuqoddm.eu-central-1.rds.amazonaws.com
Port: 3306

Networking
Availability zone: eu-central-1a
VPC: vpc-5cd60d36
Subnet group: default-vpc-5cd60d36
Subnets: subnet-c376bb8f, subnet-0552ac79, subnet-b06deeda

Security
VPC security groups: default (sg-7e2c8106) (active)
Public accessibility: Yes
Certificate authority: rds-ca-2019
Certificate authority date: Aug 22nd, 2024

Figure 2-3 Online AWS RDS dashboard

2.2.2 Back-end + API

2.2.2.1 Flask

Flask[7] is one of the two most popular Python-based web-development framework: Flask and Django. Both are open source and well documented frameworks, though Flask is chosen over Django in this particular project: Flask is considered a “micro-framework” compared to Django: Flask has no built-in admin interface, no built-in lightweight CRUD operation supports, no ORM (Object Relational Mapping) out of the box, unlike Django which has all of these features and many others pre-packaged. To add extra functionalities into Flask, additional plugins must be manually installed.

This makes Flask very simple to setup and considering that this backend only needs two main extra plugins (pandas and SQLAlchemy), Flask is preferable to the more cumbersome Django in this case. In this project, Flask is used in the backend application to setup a REST API and provide endpoints for the frontend application to hook into.

2.2.2.2 Pandas

Pandas[8] is an open source Python library for data analysis and manipulation. Pandas works by organizing data into DataFrames: 2 dimensional “table” structure that is not unlike a CSV table. Pandas is good for reading incoming CSV files and using them to build DataFrames, as well as exporting compiled DataFrames to other formats, in our case JSON for the API.

In this application, pandas is used mainly for loading the raw CSV files into the application as DataFrames, performing ETL operations on those DataFrames to load data into the database, and transforming DataFrames into JSON responses for GET requests.

2.2.2.3 SQLAlchemy

[7] Flask’s documentation can be found here: https://flask.palletsprojects.com/en/1.1.x/
[8] pandas’s documentation can be found here: https://pandas.pydata.org/docs/

SQLAlchemy[9] is also an open source Python library that provides SQL/ORM features. It supports SQLite, PostgreSQL, **MySQL**, Oracle, MS-SQL, Firebird, Sybase and others.

SQLAlchemy interacts with the database using SQL standards, but the backend application, data entities are mapped to objects, and no actual hard coded SQL queries were needed.

In this application, SQLAlchemy is used to define the database schema, and perform read-write operations on the MySQL database. This application uses Flask-SQLAlchemy[10], a specific version of SQLAlchemy which is packed as a Flask extension.

2.2.3 Front-end

2.2.3.1 React

React[11] is a popular JavaScript library for building front-end applications. It works by wrapping GUI elements in Components using JSX syntax and exposing their state and props (custom components can also be defined). Variables are loaded into these components by manipulating their state and props. React applications are made by assembling these Components together.

The library comes with a set of premade components, but extra ones can be added via installing third-party packages.

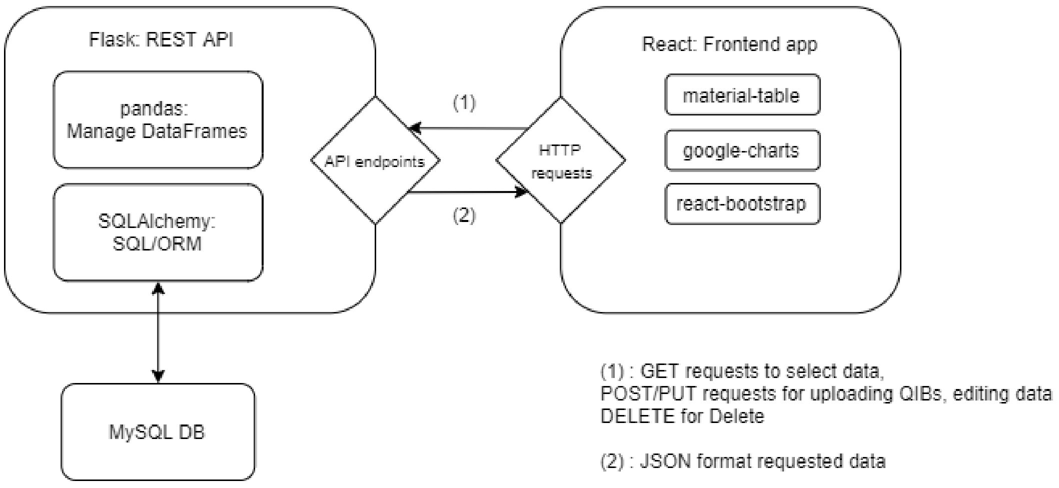
The 3 main packages used in this application are:

- React Bootstrap[12]: A library of premade components that are styled using Bootstrap.
- Material-table[13]: a flexible Table component that was based on MaterialUI (another styling library). The table supports column sorting, cell search, CSV exports, and exposes hooks for custom functions.
- React-google-charts[14]: for visualizing features into scatterplots. React-google-charts is easy to setup and use, with the caveat that the application must be online, since the chart is rendered from Google's server.

2.2.3 Architecture of project

Based on the solution overview from section 2.1.3, the chosen technologies are organized as below:

[9] SQLAlchemy's documentation can be found here: https://docs.sqlalchemy.org/en/13/
[10] Flask-SQLAlchemy's documentation can be found here: https://flask-sqlalchemy.palletsprojects.com/en/2.x/
[11] React's main page can found here: https://reactjs.org
[12] React Bootstrap's documentation can be found here: https://react-bootstrap.github.io
[13] Material-table's documentation can be found here: https://material-table.com/#/
[14] React-google-chart's documentation can be found here: https://react-google-charts.com


Flask: REST API
pandas: Manage DataFrames
SQLAlchemy: SQL/ORM
MySQL DB
API endpoints
HTTP requests
(1)
(2)
React: Frontend app
material-table
google-charts
react-bootstrap
(1) : GET requests to select data, POST/PUT requests for uploading QIBs, editing data DELETE for Delete
(2) : JSON format requested data
Figure 2-4 Detailed project architecture

2.2.5 Technologies comparison

Compared to i2b2, which is a popular solution for managing biomedical data, this project is specifically built for managing radiomics feature sets, as opposed to general patient information. Moreover, since i2b2 comes as a package, it is quite rigid in implementation: a mobile client would be hard to implement, for example.

Comparison table	**i2b2**	**Flask + Pandas + SQLAlchemy + React**
Definition	Data model + server + interface package for medical analytics	Data model + server + interface to manage feature sets
Opensource	yes	All dependent libraries are opensource
Focus	Biomedical data, clinical patient info	Quantitative Image Biomarkers
Technology stack needed	Backend: Java, JBoss, Apache, xml Frontend: IIS, PHP	Backend: Python, Flask Frontend: React, JavaScript
Data model	Based on star schema	Generic relational data schema
GUI	yes	yes
DB Support	Oracle, PostgreSQL, SQL Server	(SQLAlchemy) SQLite, PostgreSQL, **MySQL**, Oracle, MS-SQL, Firebird, Sybase and others

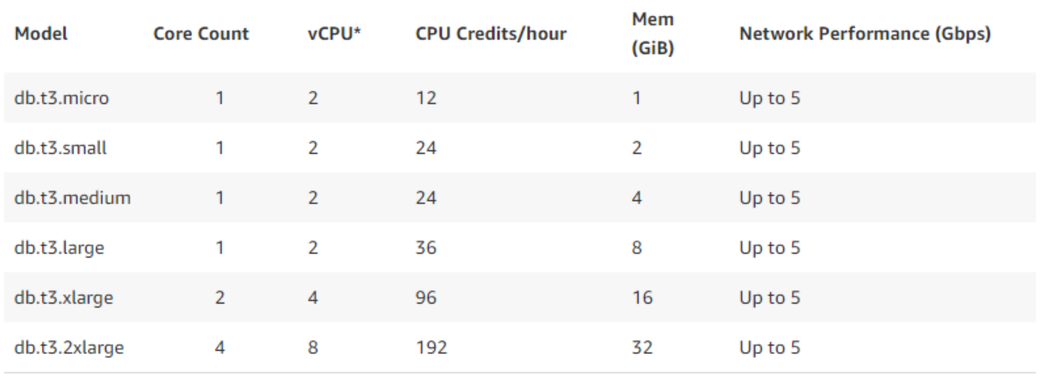
Figure 2-5 Comparison table of i2b2 and this application technologies

2.3 Tools used

2.3.1 Amazon Relational Database Service

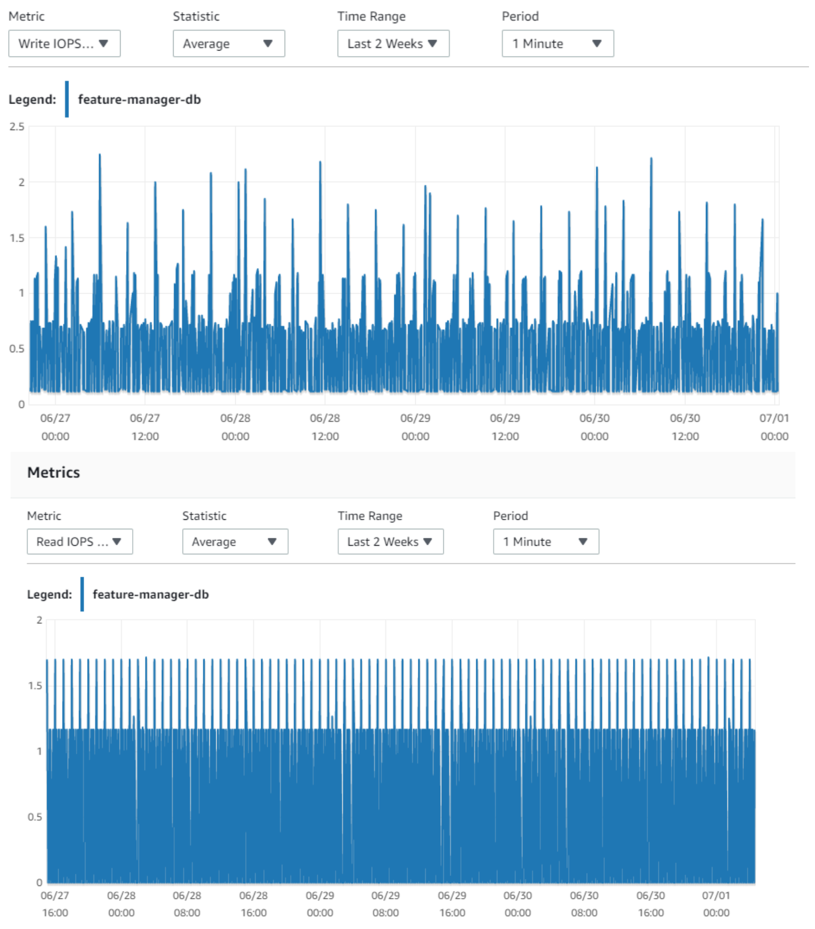
Amazon Web Services is a subsidiary of Amazon that provides a range of cloud storage solutions, including hosting databases. The service is called Amazon Relational Database Service (Amazon RDS). For its free-tier[15], Amazon RDS offers 750 hours of burstable db.t2.micro instances.

[15] Information about Amazon RDS's free tier can be found here: https://aws.amazon.com/rds/free/



Model	Core Count	vCPU*	CPU Credits/hour	Mem (GiB)	Network Performance (Gbps)
db.t3.micro	1	2	12	1	Up to 5
db.t3.small	1	2	24	2	Up to 5
db.t3.medium	1	2	24	4	Up to 5
db.t3.large	1	2	36	8	Up to 5
db.t3.xlarge	2	4	96	16	Up to 5
db.t3.2xlarge	4	8	192	32	Up to 5

Figure 2-6 db.t2.micro specifications compared to other general instances



Metric: Write IOPS... Statistic: Average Time Range: Last 2 Weeks Period: 1 Minute

Legend: feature-manager-db

Metrics

Metric: Read IOPS ... Statistic: Average Time Range: Last 2 Weeks Period: 1 Minute

Legend: feature-manager-db

Figure 2-7 Write/Read IOP (input/output operations per second) from 27th June to 1st July

While having an always online database was convenient, in practice the reading speed of this instance is noticeably slow when querying for QIBFeatures and loading them into table form. Furthermore, the database instance is designed in a way that is inconvenient to turn on and off the database server at will. The instance will autostart if left stopped for more than 7 days[16], and thus will deplete the 750 hours limit and incur charges if left unattended. Nevertheless, this demonstrates the capacity for this application to scale up if a paid tier is used.

[16] Amazon's announcement can be found here: https://aws.amazon.com/about-aws/whats-new/2017/06/amazon-rds-supports-stopping-and-starting-of-database-instances/

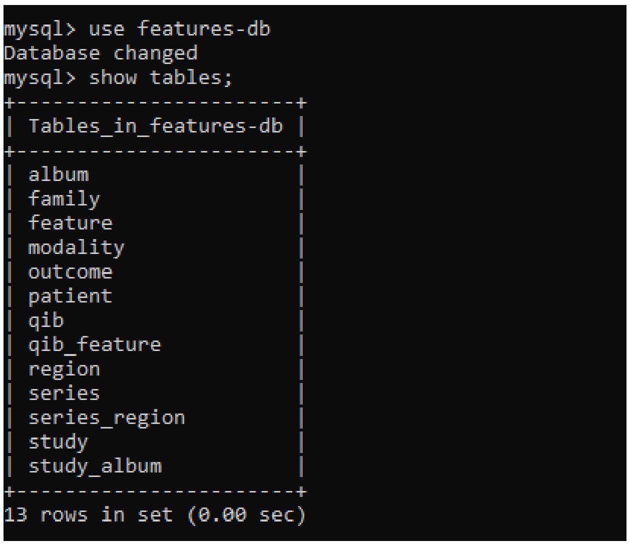
2.3.2 Local MySQL server

Slow read/write speed of AWS RDS eventually led to the implementation of a more local solution.

Setting up a local MySQL Community server on Windows is easy: simply download the installer for Windows on their official webpage and run the installer. Linux users can install from their respective distribution's repository or download and run the .deb package if installing from a computer with no connection.

For this application, MySQL Workbench is unnecessary, so only the server is needed.

The server can technically be managed through the command line client, but for convenience, DBeaver will be used as the database manager.



```
mysql> use features-db
Database changed
mysql> show tables;
+-----------------------+
| Tables_in_features-db |
+-----------------------+
| album                 |
| family                |
| feature               |
| modality              |
| outcome               |
| patient               |
| qib                   |
| qib_feature           |
| region                |
| series                |
| series_region         |
| study                 |
| study_album           |
+-----------------------+
13 rows in set (0.00 sec)
```

Figure 2-8 A table list of the application's database from the command line client

2.3.3 DBeaver

DBeaver[17] is a free open source universal database manager. This tool supports a very wide range of databases, which of course also includes MySQL. It provides the ability to connect to databases to design and view ER diagrams, monitor traffic with the dashboard, drill-down selecting/ filtering for data, among other features. For our application we will be using the Community version.

In the beginning of this project's development, this tool was used to design to database schema as well, but this process has since been moved to the backend application's models.py.

[17] DBeaver's main page can be found here: https://dbeaver.io

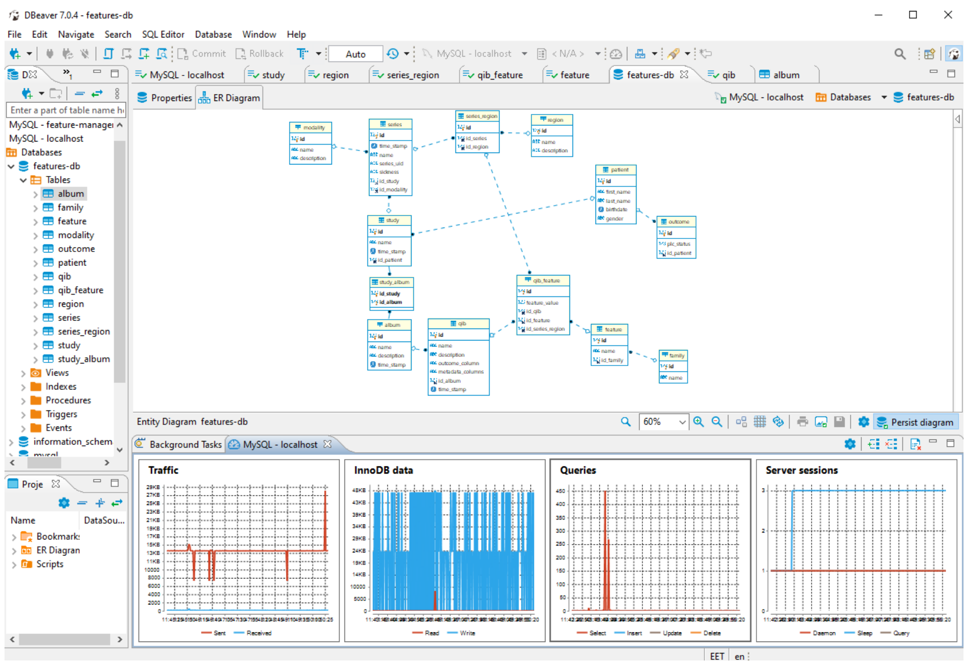


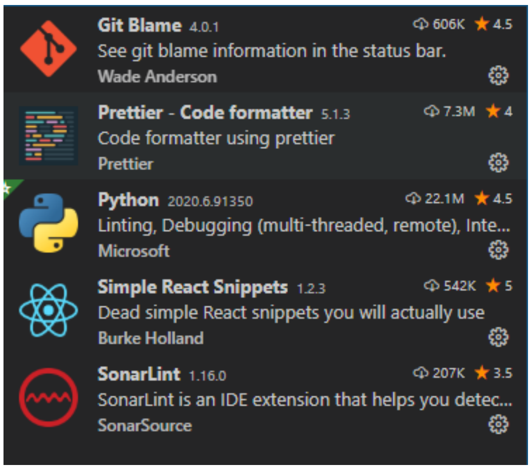
Figure 2-9 DBeaver GUI, with ER Diagram view and Dashboard enabled

2.3.4 Visual Studio Code

Visual Studio Code[18] is a cross platform code IDE with robust features. For this application's use, it supports linting for Python and JavaScript.

VSCode's source code repository is open source, but the software itself is not (Dias, 2015), since it ships with some extra telemetry and branding features from Microsoft. Those who prefer a more open source IDE could opt for VSCodium[19] which is built directly from the open source codebase and thus contains no telemetry.

Both versions can be further extended by installing extensions, which provide extra functionalities such as code formatting, opinionated linting (identify 'dirty' code), version control, and so on.


Git Blame 4.0.1 606K ★ 4.5
See git blame information in the status bar.
Wade Anderson

Prettier - Code formatter 5.1.3 7.3M ★ 4
Code formatter using prettier
Prettier

Python 2020.6.91350 22.1M ★ 4.5
Linting, Debugging (multi-threaded, remote), Inte...
Microsoft

Simple React Snippets 1.2.3 542K ★ 5
Dead simple React snippets you will actually use
Burke Holland

SonarLint 1.16.0 207K ★ 3.5
SonarLint is an IDE extension that helps you detec...
SonarSource

Figure 2-10 List of extensions used over the course of this project's development

[18] VSCode's main page can be found here: https://code.visualstudio.com
[19] VSCodium's main page can be found here: https://vscodium.com

2.3.5 GitHub

GitHub is a Git-based online service for hosting project. Code for both the back-end and front-end application are hosted on GitHub:

- Back-end: https://github.com/genttunn/python-rest-api.git
- Front-end: https://github.com/genttunn/feature-manager

3. RESULTS

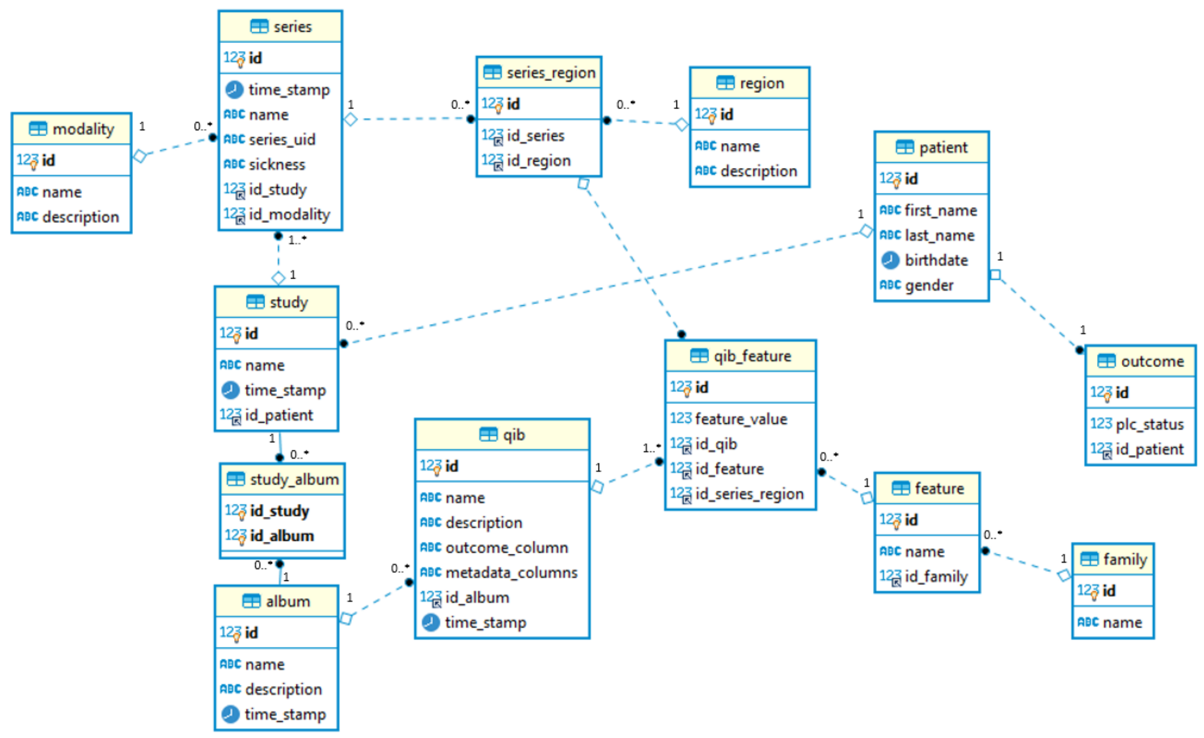
This chapter will present the result of the project. The explanation is structured by application layer: database, back-end, front-end.

3.1 Database

This section and the next will explain how data is broken down into entities in the database.

3.1.1 Data model

This section explains the result relational data model used to store metadata. Some entities come directly from the terminologies mentioned in section 2.1.1, some others are for enforcing many-to-many relationships between the entities.



From left to right:

- **modality:** Refers to the imaging modality used in a particular series.
- **series:** Each series contains a reference to its modality and region(s) of interest. One series can have multiple regions of interest, and a region of interest can be referenced by multiple series.
- **series_region:** Table to represent Series and Region of Interest's many-to-many relationship.
- **region:** Region of Interest. Contains region name and description.
- **study:** A study can contain many series. However, a study can only refer to one patient. A study can be included in multiple albums, so the relationship between Album and Study is many-to-many.
- **album:** Features from one album can be extracted multiple times, depending on the criteria, so one Album can be referenced by many QIBs. Each QIB can only reference one album.
- **study_album:** Represents Study and Album's many-to-many relationship.
- **patient:** Each Patient entry contains name, description, birthdate and gender.

- **outcome:** Each outcome refers to a patient, and plc_status of the outcome is binary (0/1).
- **qib:** Short for Quantitative Image Biomarkers. Deleting a QIB will cascade delete all related qibFeatures. A QIB will be represented as a table in the front-end application.
- **qib_feature:** Represents the value cells in each CSV feature extraction. Each QIBFeature contains a reference to the QIB it belongs to, the Feature it represents, and a reference to the Series_Region table, which will give it access to the Series, Study, and related information of those entities. In the front-end application, each qib_feature entry makes up a cell in the data table.
- **feature:** Each feature belongs to one feature Family.
- **family:** Categories of feature. For this application, we use 2 types: 'texture' and 'intensity'.

3.1.2 Transforming a CSV file into database entities

To note, some of the metadata are not currently present in the extraction file (Patient birthdate/gender, Study, Family of features, etc..) because they are not necessary for the ML process.

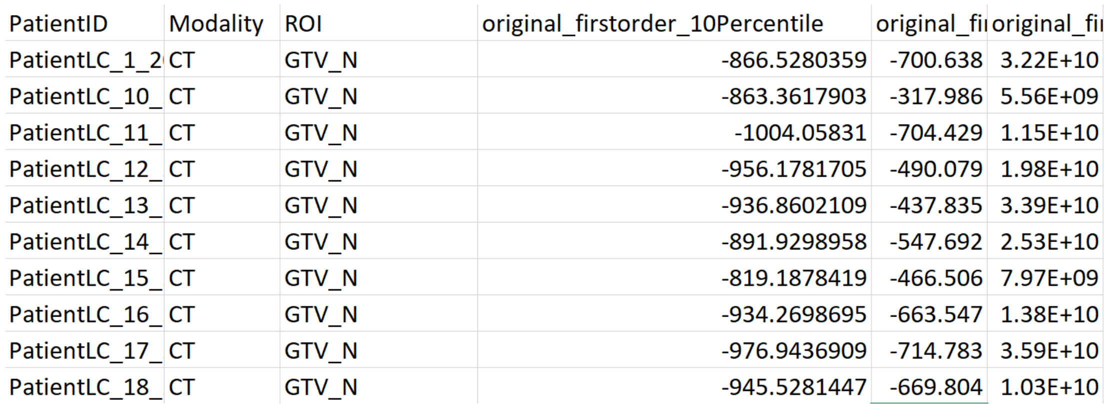
However, they are needed in this application to fill a complete data model for filtering, so the aforementioned missing metadata will be filled with mock data/ filled out by user upon uploading the file. They can also be edited later directly from the front-end application.

3.1.2.1 CSV file input

The following figures show what an extracted feature set looks like.

- PatientID: contains a number for Outcome CSV to refer to.
- Modality: modality of the feature set.
- ROI: region of interest in the feature set.
- And the rest are feature columns.

Study, series, and family of features is not given in these files, so they will be filled with generated mock data in the ETL process.

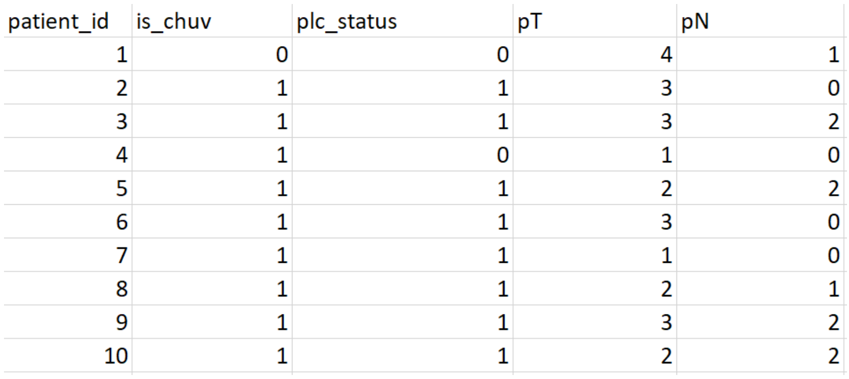


PatientID	Modality	ROI	original_firstorder_10Percentile	original_fi	original_fi
PatientLC_1_2	CT	GTV_N	-866.5280359	-700.638	3.22E+10
PatientLC_10_	CT	GTV_N	-863.3617903	-317.986	5.56E+09
PatientLC_11_	CT	GTV_N	-1004.05831	-704.429	1.15E+10
PatientLC_12_	CT	GTV_N	-956.1781705	-490.079	1.98E+10
PatientLC_13_	CT	GTV_N	-936.8602109	-437.835	3.39E+10
PatientLC_14_	CT	GTV_N	-891.9298958	-547.692	2.53E+10
PatientLC_15_	CT	GTV_N	-819.1878419	-466.506	7.97E+09
PatientLC_16_	CT	GTV_N	-934.2698695	-663.547	1.38E+10
PatientLC_17_	CT	GTV_N	-976.9436909	-714.783	3.59E+10
PatientLC_18_	CT	GTV_N	-945.5281447	-669.804	1.03E+10

Figure 3-1 Example of a feature set extraction file

For the outcome CSV, we have:

- Patient_id : refers to the number part in the main feature set.
- Plc_status: the outcome chosen as outcome column in this project.



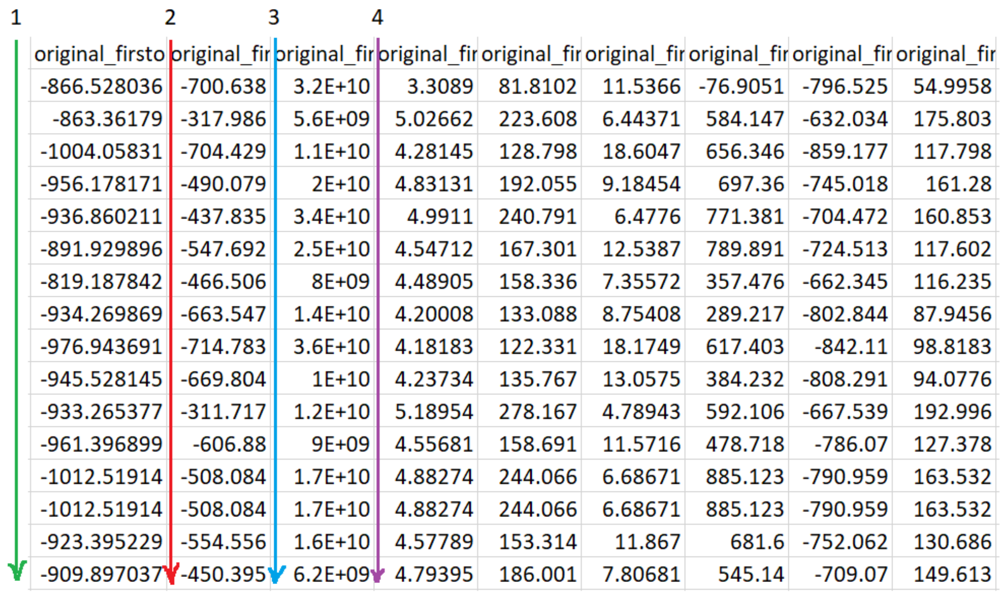
patient_id	is_chuv	plc_status	pT	pN
1	0	0	4	1
2	1	1	3	0
3	1	1	3	2
4	1	0	1	0
5	1	1	2	2
6	1	1	3	0
7	1	1	1	0
8	1	1	2	1
9	1	1	3	2
10	1	1	2	2

Figure 3-2 Outcome CSV file

3.1.2.2 Steps of loading data into the database:

1. Album name is taken from user input in the front-end application.
2. Modality and ROI are loaded using the Modality/ROI columns. Only new values that do not exist in the database are inserted.
3. Patient data is taken from the PatientID column. Currently, the number after "PatientLC_" is used as the patient number for loading outcomes into the database.
4. Along with each created Patient, 3 mock Studies will also be created for them. Series, in the ETL process, will be randomly assigned to either of these 3 mock Studies. Each newly created series will also have a new Series_Region referencing the ROI of the same row assigned to it.
5. Feature: feature names are created from the columns of the extraction file. The Family of a created Feature is randomly assigned to either 'texture' or 'family'.
6. QIBFeature (cell) entry: from each cell, the appropriate Patient (from PatientID), Study, Series, and Series_Region created from the above steps will be used to create a new QIBFeature.

3.1.3 Order of insertion



original_firsto	original_fir	original_fir	original_fir	original_fir	original_fir	original_fir	original_fir	original_fir
-866.528036	-700.638	3.2E+10	3.3089	81.8102	11.5366	-76.9051	-796.525	54.9958
-863.36179	-317.986	5.6E+09	5.02662	223.608	6.44371	584.147	-632.034	175.803
-1004.05831	-704.429	1.1E+10	4.28145	128.798	18.6047	656.346	-859.177	117.798
-956.178171	-490.079	2E+10	4.83131	192.055	9.18454	697.36	-745.018	161.28
-936.860211	-437.835	3.4E+10	4.9911	240.791	6.4776	771.381	-704.472	160.853
-891.929896	-547.692	2.5E+10	4.54712	167.301	12.5387	789.891	-724.513	117.602
-819.187842	-466.506	8E+09	4.48905	158.336	7.35572	357.476	-662.345	116.235
-934.269869	-663.547	1.4E+10	4.20008	133.088	8.75408	289.217	-802.844	87.9456
-976.943691	-714.783	3.6E+10	4.18183	122.331	18.1749	617.403	-842.11	98.8183
-945.528145	-669.804	1E+10	4.23734	135.767	13.0575	384.232	-808.291	94.0776
-933.265377	-311.717	1.2E+10	5.18954	278.167	4.78943	592.106	-667.539	192.996
-961.396899	-606.88	9E+09	4.55681	158.691	11.5716	478.718	-786.07	127.378
-1012.51914	-508.084	1.7E+10	4.88274	244.066	6.68671	885.123	-790.959	163.532
-1012.51914	-508.084	1.7E+10	4.88274	244.066	6.68671	885.123	-790.959	163.532
-923.395229	-554.556	1.6E+10	4.57789	153.314	11.867	681.6	-752.062	130.686
-909.897037	-450.395	6.2E+09	4.79395	186.001	7.80681	545.14	-709.07	149.613

Figure 3-3 Order of inserting QIBFeature

The order of looping over the cells is: column by column, from left to right, top to bottom. For example, in figure 3-3, the order would be 1 -> 2 -> 3 -> 4. The reason for the looping order to be column-based is because each column represents the complete data of one feature in the set. Going by column allows for a batch insertion of that feature's QIBFeatures (cells) in one loop. A row-based looping order would mean switching feature every cell, and thus is less efficient.

The order of inserting QIBFeature and query for them is the same, so upon reloading data from the database back into table form, the rows are accurate.

3.2 Back-end

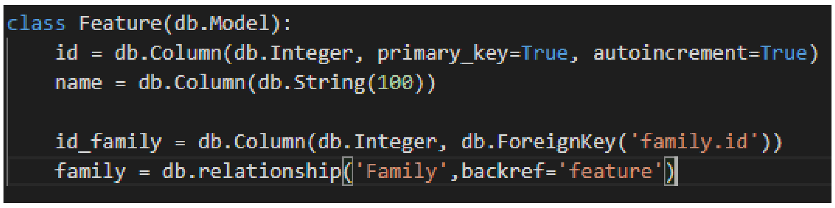
3.2.1 Defining the database schema with SQLAlchemy

The process of defining models for the database can be done entirely with SQLAlchemy by defining classes that correspond to database entities.

Naming convention: Class names in the backend application are named using PascalCase, but without explicitly stating table names, PascalCase classes will be converted to snake_case table names in the database.

To set up SQLAlchemy: db = SQLAlchemy(app). SQLAlchemy variables and functions are then accessible through db.

- To define a class (entity) and set foreign keys:

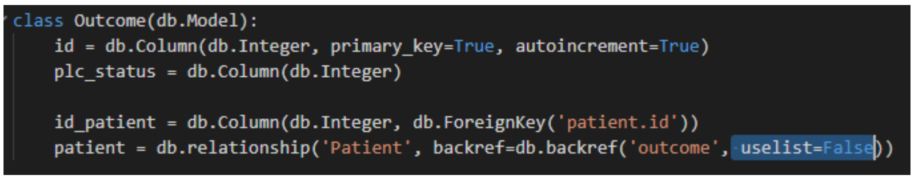


```
class Feature(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(100))

    id_family = db.Column(db.Integer, db.ForeignKey('family.id'))
    family = db.relationship('Family',backref='feature')
```

Figure 3-4 Defining Feature table

Since SQLAlchemy supports ORM, a Feature's Family can be accessed with the Feature.family property.

- For one-to-one relationships:



```
class Outcome(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    plc_status = db.Column(db.Integer)

    id_patient = db.Column(db.Integer, db.ForeignKey('patient.id'))
    patient = db.relationship('Patient', backref=db.backref('outcome', uselist=False))
```

Figure 3-5 Defining Outcome table

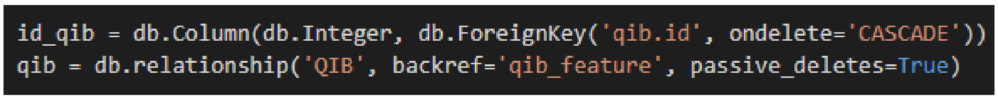
Each Patient has only one Outcome, and vice versa (in this project scope, only plc_status is used as Outcome). To denote that relationship, simply add "uselist = False" when defining the foreign key.

- Cascading:

Default cascading behaviour is only enabled for save-update and merge (so generally UPDATE queries). To enable them for delete there are 2 ways:

- Through SQLAlchemy 'delete' properties:
  parent = relationship('Parent', backref=backref('children', cascade='all,delete'))

- Or through the database ON DELETE, which is 'vastly more efficient' (Documentation, n.d.) than the former. This method is used in the application. To set this up from the child class, in this case QIBFeature which will cascade delete if its QIB is deleted:



```
id_qib = db.Column(db.Integer, db.ForeignKey('qib.id', ondelete='CASCADE'))
qib = db.relationship('QIB', backref='qib_feature', passive_deletes=True)
```

Figure 3-6 Defining QIBFeature relationships

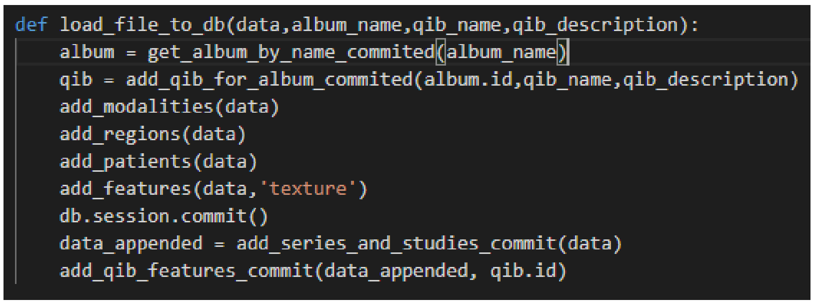
3.2.2 ETL process

As discussed in section 3.1.2, some metadata from Kheops were not carried over to the extracted feature set .CSV, and mock data/ default values will be used to fill the database. Those metadata are:

- Study: for each Patient, 3 random studies will be created. They then are chosen at random for each patient to fill in the Study table.
- Series: series.name property will be from PatientID column, and each will be given a random series_uid number.
- Family: two families will be created ('texture' and 'intensity'). New features are automatically assigned to 'texture'. Family of feature can be edited manually later from the GUI.
- Patient: patient.last_name will be used as a number field for inserting Outcomes (explained more in section 3.2.2.1). Gender is by default Female (F), and birthdate as '2000-01-01'.

There are 2 main kinds of ETL process needed: one to load raw feature sets into the database, and for saving custom QIB selections back into the database, due to them having different column structure. Another ETL sub-process is also needed for loading outcomes of patients into the database, because the outcome list is a separate CSV file.

3.2.2.1 Loading raw feature sets

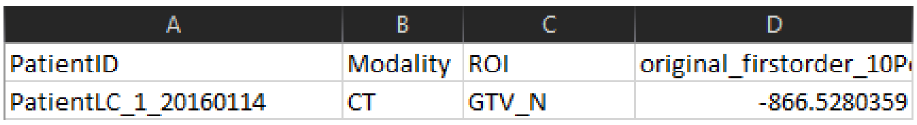


```
def load_file_to_db(data,album_name,qib_name,qib_description):
    album = get_album_by_name_commited(album_name)
    qib = add_qib_for_album_commited(album.id,qib_name,qib_description)
    add_modalities(data)
    add_regions(data)
    add_patients(data)
    add_features(data,'texture')
    db.session.commit()
    data_appended = add_series_and_studies_commit(data)
    add_qib_features_commit(data_appended, qib.id)
```

Figure 3-7 Steps of loading raw QIBs

To load a QIB into the database, the user will need to provide:

- The CSV file with appropriate columns: PatientID, Modality, ROI, and feature columns. PatientID must contain a number after PatientLC_ for the outcome list to refer to.
- Name of the album this QIB belongs to: if the name is new then a new album will be created.
- QIB name and description.



A	B	C	D
PatientID	Modality	ROI	original_firstorder_10P
PatientLC_1_20160114	CT	GTV_N	-866.5280359

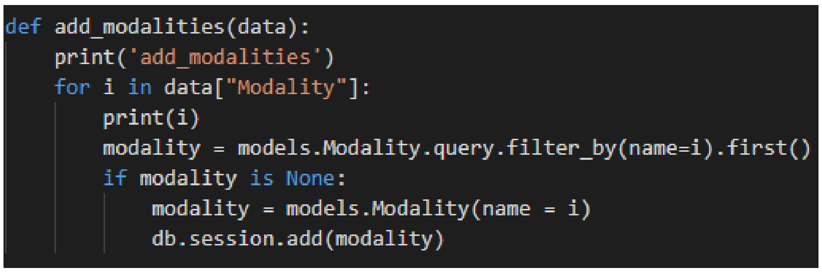
Figure 3-8 A feature set with valid columns

The main steps in adding a raw feature set are:

- Loading the CSV file into our back-end application as a DataFrame.
- Creating a new QIB entry.
- Loading metadata (Album, Modality, Region of Interest, Patient) relevant to the created QIB.
- Generating Studies/Series/Series_region data and append the newly generated Series_region as a column into the DataFrame.
- With the Series_region appended, add the DataFrame's cell data into our database as QIBFeature. Thanks to Series_region, from each QIBFeature its relevant Region, Series and by extent, Series.study, Study.patient can be accessed.
- After this the sub-process of loading the outcome list can be started, to load outcome for each patient into the database's Outcome table.

Detailed steps of the ETL process to clean and load a feature set into the database:

1. Find the user's defined album by input name, or create one with that name.
2. Create a new QIB entry and set it to reference that album.
3. Loop through Modality and ROI columns to and add any new entry into the database (existing ones are ignored).



```
def add_modalities(data):
    print('add_modalities')
    for i in data["Modality"]:
        print(i)
        modality = models.Modality.query.filter_by(name=i).first()
        if modality is None:
            modality = models.Modality(name = i)
            db.session.add(modality)
```

Figure 3-9 Adding modalities

4. Add patient: With the current batch of extracted feature sets, the format of PatientID column is like this:

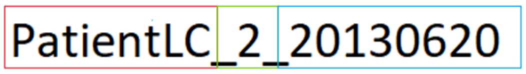
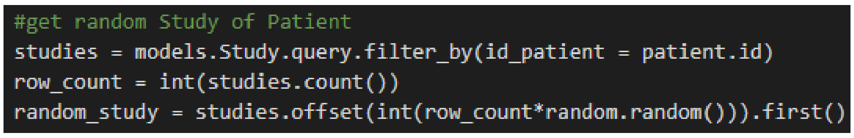


Figure 3-10 Red: hardcoded, Green: used as patient id in Outcome table, Blue: extraction date, not used

   The red part will be used as the patient first name, green part as last name (a temporary solution to load outcome tables). A quirk of this project's dataset is that the green part is used as reference for Outcome data. Blue part is not relevant to the data and is ignored. Once the outcomes are loaded into the table and linked to their respective patients, the patient name can be changed.
5. Upon inserting a new patient, **3 mock studies** will be created referencing that patient. This is a temporary solution to address the lack of studies metadata from the CSV.
6. Features are now added in the same way as Modality and ROI, but instead of looping over their respective columns (vertical loop through rows), all column headers except for metadata ones are looped through (horizontal loop through columns). The family of the feature is left on default as 'texture'.
7. Adding series into studies: This is one of the more difficult part of the ETL process as it relies on mock data (Studies). The goal here is to loop through each row and append a new column

Series_Region into the DataFrame, which would give the QIBFeature a reference to its study and its region. To do this, we need to append an empty Series_Region column to the DataFrame.
The first step in this part is to get the Patient of the row, this SELECT query should never turn up empty since the Patient is already inserted into the database. After which, a random study among the 3 belonging to that Patient is selected.

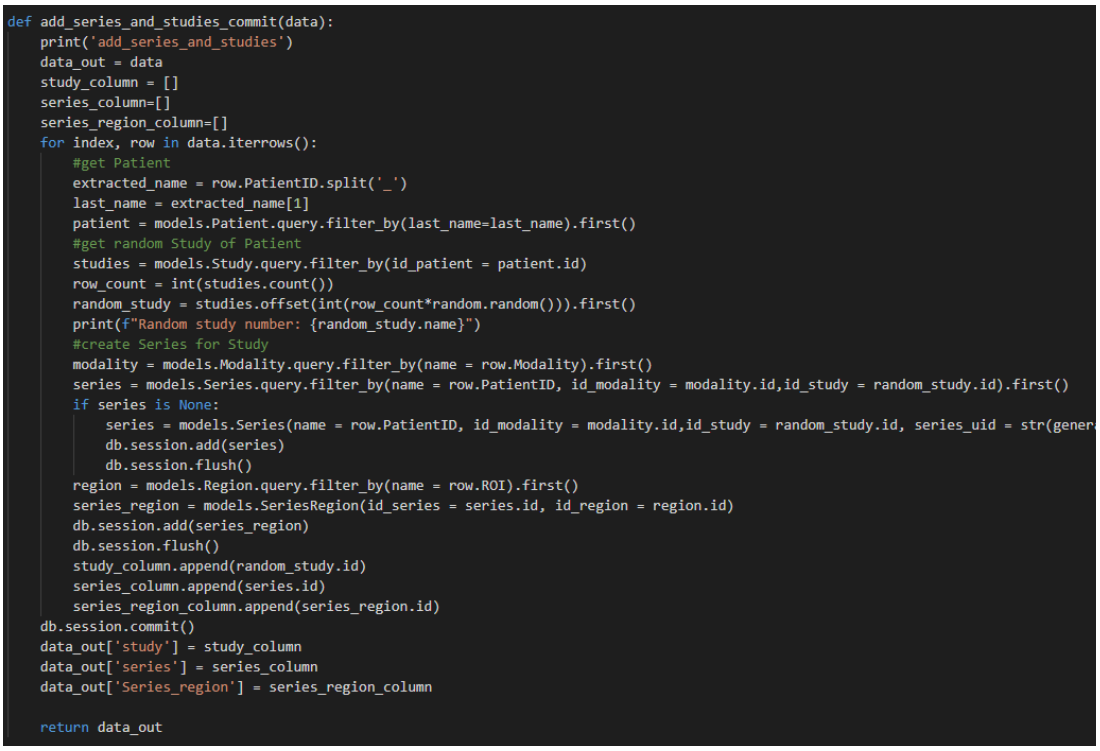


```
#get random Study of Patient
studies = models.Study.query.filter_by(id_patient = patient.id)
row_count = int(studies.count())
random_study = studies.offset(int(row_count*random.random())).first()
```

Figure 3-11 Random selection using offset

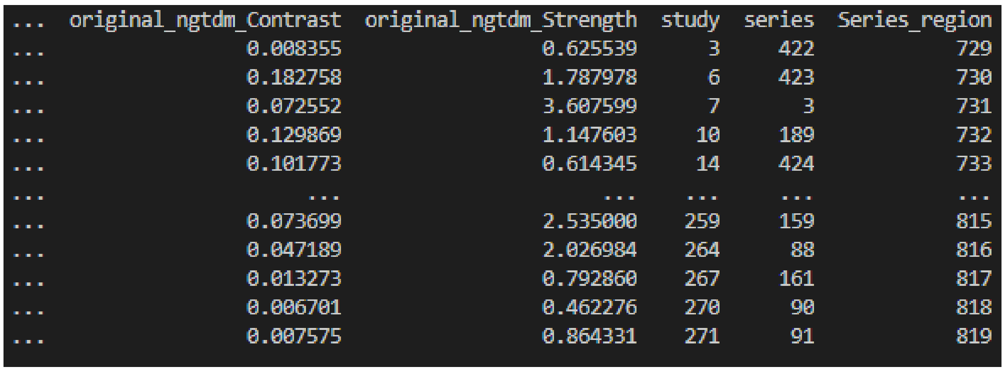
With our Study in hand, either a new Series or an existing one with the same name, modality, and study will be created (or selected). With this Series, we simply need to find that row's ROI, create a Series_Region entry with that Series and Region, and append the newly created SeriesRegion id into our Series_region column.

The result of this step is returned in DataFrame format to be used in the next step.



```
def add_series_and_studies_commit(data):
    print('add_series_and_studies')
    data_out = data
    study_column = []
    series_column=[]
    series_region_column=[]
    for index, row in data.iterrows():
        #get Patient
        extracted_name = row.PatientID.split('_')
        last_name = extracted_name[1]
        patient = models.Patient.query.filter_by(last_name=last_name).first()
        #get random Study of Patient
        studies = models.Study.query.filter_by(id_patient = patient.id)
        row_count = int(studies.count())
        random_study = studies.offset(int(row_count*random.random())).first()
        print(f"Random study number: {random_study.name}")
        #create Series for Study
        modality = models.Modality.query.filter_by(name = row.Modality).first()
        series = models.Series.query.filter_by(name = row.PatientID, id_modality = modality.id,id_study = random_study.id).first()
        if series is None:
            series = models.Series(name = row.PatientID, id_modality = modality.id,id_study = random_study.id, series_uid = str(gener
            db.session.add(series)
            db.session.flush()
        region = models.Region.query.filter_by(name = row.ROI).first()
        series_region = models.SeriesRegion(id_series = series.id, id_region = region.id)
        db.session.add(series_region)
        db.session.flush()
        study_column.append(random_study.id)
        series_column.append(series.id)
        series_region_column.append(series_region.id)
    db.session.commit()
    data_out['study'] = study_column
    data_out['series'] = series_column
    data_out['Series_region'] = series_region_column

    return data_out
```

Figure 3-12 Linking Series, Studies, and Region



```
...  original_ngtdm_Contrast  original_ngtdm_Strength  study  series  Series_region
...                 0.008355                 0.625539      3     422            729
...                 0.182758                 1.787978      6     423            730
...                 0.072552                 3.607599      7       3            731
...                 0.129869                 1.147603     10     189            732
...                 0.101773                 0.614345     14     424            733
...                      ...                      ...    ...     ...            ...
...                 0.073699                 2.535000    259     159            815
...                 0.047189                 2.026984    264      88            816
...                 0.013273                 0.792860    267     161            817
...                 0.006701                 0.462276    270      90            818
...                 0.007575                 0.864331    271      91            819
```

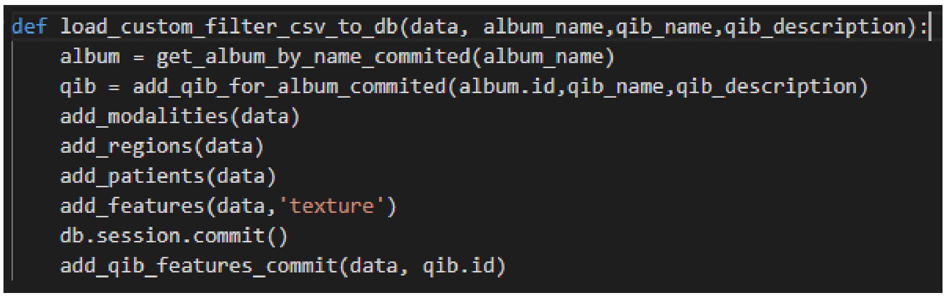
Figure 3-13 Appended DataFrame with new columns

8. Adding QIBFeature: we now have enough metadata to actually load individual cell values into the database. With the appended DataFrame from the previous step, we loop over the table from left to right, top to bottom and insert cell values as QIBFeature if the column is a Feature column. Each QIBFeature will have a value, the QIB it belongs to, the Feature it belongs to, and a reference to a SeriesRegion which gives it access to its Study and ROI.
9. The outcome list is presented as a separate CSV file that contain outcome variables, and patient id, which is not the same as our database's generated id. The id number is found in the PatientID column of the main CSV, right after "PatientLC_". Loading them into the database is a straightforward task of filtering for the relevant patient and assigning that outcome to them. For our application, we will now only take the plc_status column.

3.2.2.2 Loading custom QIB

The frontend application provides a dynamic table that can be filtered, sorted, and selectively exported. To save that custom QIB back into the database the user will need to provide:

- The custom exported CSV with appropriate column names (that are automatically set with the export): PatientName, plc_status, Modality, ROI, Series_region, and features columns.
- Album name: users can save the custom QIB to any album, but is preferable to save them in one custom_qibs album



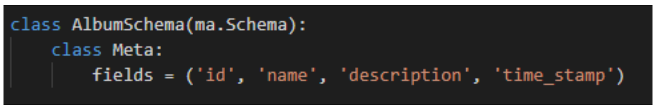
```python
def load_custom_filter_csv_to_db(data, album_name,qib_name,qib_description):
    album = get_album_by_name_commited(album_name)
    qib = add_qib_for_album_commited(album.id,qib_name,qib_description)
    add_modalities(data)
    add_regions(data)
    add_patients(data)
    add_features(data,'texture')
    db.session.commit()
    add_qib_features_commit(data, qib.id)
```

Figure 3-14 Loading custom QIBs

The process is very much similar to loading a raw feature set, with one difference being no extra series or studies is needed, and the table already comes with an appended Series_region column.

3.2.3 REST API endpoints

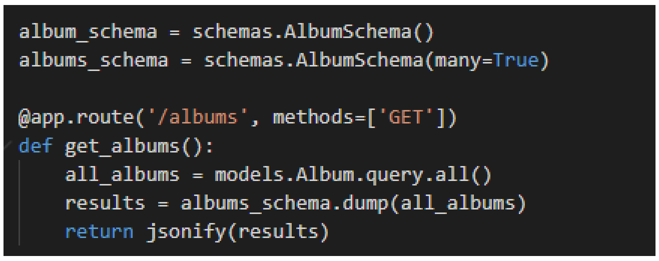
One more python library was installed to make the API work: flask-marshmallow[20]. Flask-marshmallow is a object serialization/deserialization tool: it is used to deserialize objects so we could return it as JSON format. To use flask-marshmallow, we first need to define Schemas, with field names similar to the model's properties:



```
class AlbumSchema(ma.Schema):
    class Meta:
        fields = ('id', 'name', 'description', 'time_stamp')
```

Figure 3-15 Schema of Album

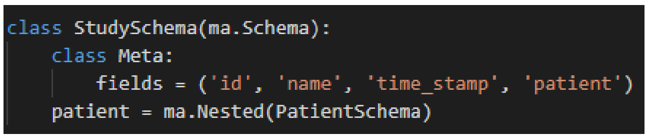
Then we can use it to dump Album into AlbumSchema and return it as JSON:



```
album_schema = schemas.AlbumSchema()
albums_schema = schemas.AlbumSchema(many=True)

@app.route('/albums', methods=['GET'])
def get_albums():
    all_albums = models.Album.query.all()
    results = albums_schema.dump(all_albums)
    return jsonify(results)
```

Figure 3-16 GET request of all albums

flask-marshmallow also supports nested schemas. The schema being nested needs to be defined beforehand.



```
class StudySchema(ma.Schema):
    class Meta:
        fields = ('id', 'name', 'time_stamp', 'patient')
    patient = ma.Nested(PatientSchema)
```

Figure 3-17 Nested Patient schema inside Study

The API exposes the following endpoints:

Method	**Endpoint**	**Description**
GET	/albums	Return list of all albums
POST	/albums	Add new album
PUT	/albums/<album_id>	Edit name and description of album
DELETE	/albums/<album_id>	Delete album (only albums with no qibs referred)
GET	/patients	Return list of all patients
PUT	/patients/<patient_id>	Edit patient information (*)
GET	/modalities	Return list of all modalities
PUT	/modalities/<modality_id>	Edit name and description of modality
GET	/regions	Return list of all regions
PUT	/regions/<region_id>	Edit name and description of region
GET	/families_features	Return list of all feature families. Each family has a nested list of its features.
PUT	/features/<feature_id>	Edit name and description of feature

[20] flask-marshmallow 's documentation: https://flask-marshmallow.readthedocs.io/en/latest/

POST	/families	Add new family
DELETE	/families/<family_name>	Delete family (only those with no features)
GET	/features/<qib_id>	Return list of features in a particular QIB
GET	/qibs	Return all QIBs
GET	/qibs?album=<album_id>	Return list of QIBs of a particular album
GET	/qibs?date=<date_string>	Return list of QIBs created *after* input date
PUT	/qib/tag/outcome/<qib_id>	Edit outcome_column of a QIB, return updated QIB
PUT	/qib/<qib_id>	Edit a QIB's name and description
DELETE	/qib/<qib_id>	Delete a QIB
GET	/qib_features /<qib_id>	Return all QIBFeatures of a QIB (**)
GET	/statistics	Return current count of current series, studies, patients, QIBs in database
GET	/chart/scatterplot/<qib_id> /<feature_1>/<feature_2	Return a list of QIBFeature values of 2 features of a QIB, and their outcome status (***)

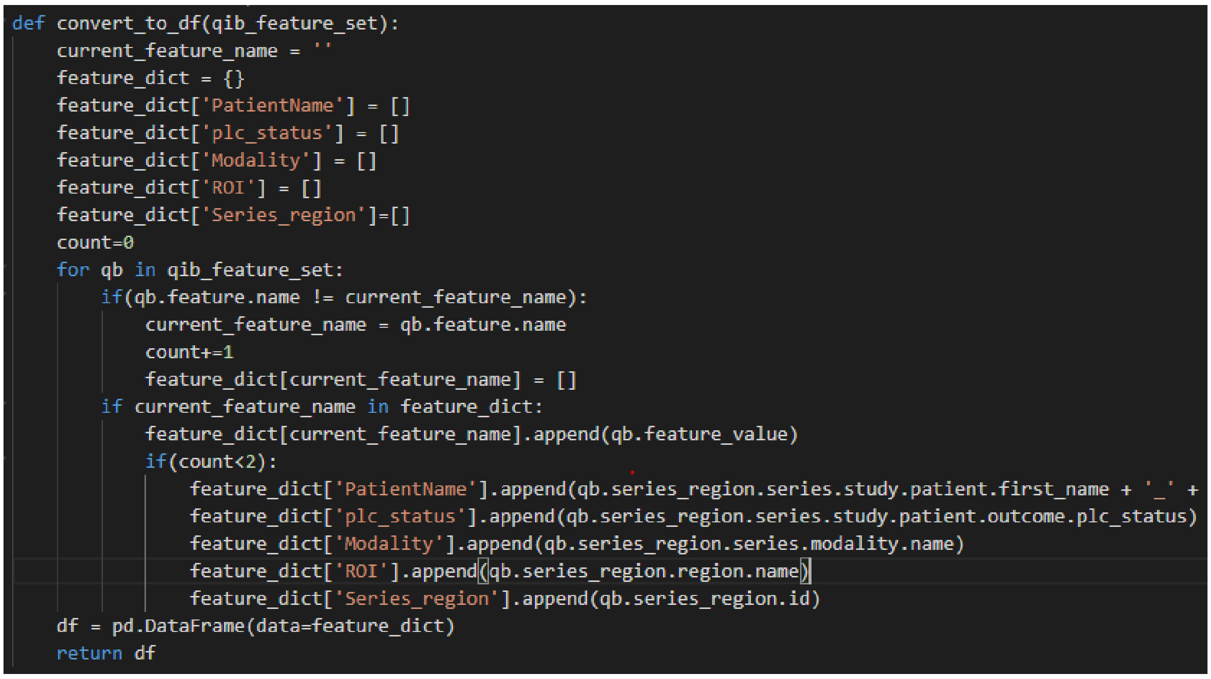
Figure 3-18 Table of API's endpoints

3.2.3.1 Editing patient information (*)

Users can edit patient's first name, birthdate, gender, and plc_status. Last name is read-only, since new Outcome CSV files rely on the last_name number as a reference (section 3.2.2.1)

3.2.3.2 Converting list of QIBFeatures into a table (**)

By order of insertion (right-to-left, top-to-bottom of table), QIBFeatures in the database are always sorted by the same order, and we can leverage that order to turn the list of QIBFeatures back into table form, and also add extra columns. First, the list of QIBFeatures belonging to a QIB is queried, then it is put through a converter to turn it back into a DataFrame:

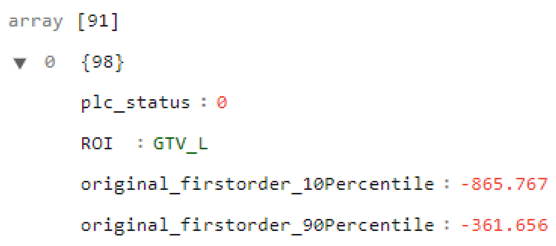


```
def convert_to_df(qib_feature_set):
    current_feature_name = ''
    feature_dict = {}
    feature_dict['PatientName'] = []
    feature_dict['plc_status'] = []
    feature_dict['Modality'] = []
    feature_dict['ROI'] = []
    feature_dict['Series_region']=[]
    count=0
    for qb in qib_feature_set:
        if(qb.feature.name != current_feature_name):
            current_feature_name = qb.feature.name
            count+=1
            feature_dict[current_feature_name] = []
        if current_feature_name in feature_dict:
            feature_dict[current_feature_name].append(qb.feature_value)
            if(count<2):
                feature_dict['PatientName'].append(qb.series_region.series.study.patient.first_name + '_' +
                feature_dict['plc_status'].append(qb.series_region.series.study.patient.outcome.plc_status)
                feature_dict['Modality'].append(qb.series_region.series.modality.name)
                feature_dict['ROI'].append(qb.series_region.region.name)
                feature_dict['Series_region'].append(qb.series_region.id)
    df = pd.DataFrame(data=feature_dict)
    return df
```

Figure 3-19 Converting list of QIBFeatures to Table

Explanation: first we create an empty dictionary and define the metadata/outcome keys. Then the list of QIBFeature is looped through, and every time QIBFeature.feature changes (including the first time), a new key is made. If QIBFeature.feature is the same, then QIBFeature.feature_value is appended to that key. For the first iteration only, metadata and outcome column values will also be appended to their respective keys, to prevent duplicates of these columns leading to uneven column lengths. Finally, the dictionary is converted into a DataFrame (column lengths must be the same) and returned in JSON with 'records' orientation.

Sample JSON result: Array of 91 objects, each object has 98 key-value pairs -> Table of 91 rows and 98 columns.

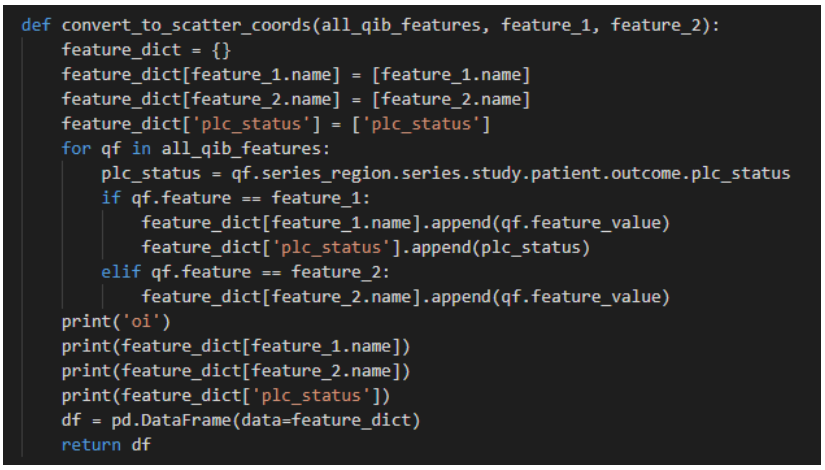


```
array [91]
▼ 0  {98}
      plc_status : 0
      ROI  : GTV_L
      original_firstorder_10Percentile : -865.767
      original_firstorder_90Percentile : -361.656
```

Figure 3-20 Tree view of a table JSON

3.2.3.3 Converting list of QIBFeatures into a scatterplot (***)

The goal is to create a bivariate scatterplot of 2 features, based on their outcome value. For this application, plc_status is used as the default outcome value. First, a list of QIBFeatures from 2 selected features are queried from the database. Due to the order of insertion (left-to-right, top-to-bottom), this list will be in the same order as though we select 2 columns from the CSV table.

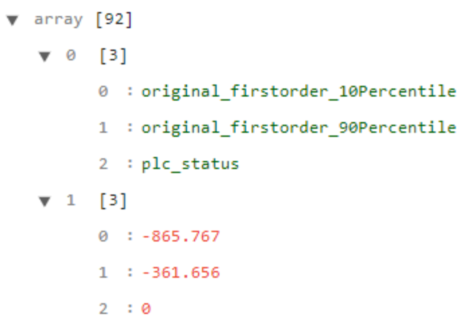


```python
def convert_to_scatter_coords(all_qib_features, feature_1, feature_2):
    feature_dict = {}
    feature_dict[feature_1.name] = [feature_1.name]
    feature_dict[feature_2.name] = [feature_2.name]
    feature_dict['plc_status'] = ['plc_status']
    for qf in all_qib_features:
        plc_status = qf.series_region.series.study.patient.outcome.plc_status
        if qf.feature == feature_1:
            feature_dict[feature_1.name].append(qf.feature_value)
            feature_dict['plc_status'].append(plc_status)
        elif qf.feature == feature_2:
            feature_dict[feature_2.name].append(qf.feature_value)
    print('oi')
    print(feature_dict[feature_1.name])
    print(feature_dict[feature_2.name])
    print(feature_dict['plc_status'])
    df = pd.DataFrame(data=feature_dict)
    return df
```

Figure 3-21 Converting QIBFeatures into scatterplot data

With the QIBFeatures selected, an array of arrays is created containing the feature values, as well as their patient's plc_status (that we could access from a QIBFeature.series_region).

Sample JSON result: Array of 92 arrays, with 1 being the column names, and 91 data points.



```
▼ array [92]
  ▼ 0  [3]
        0  : original_firstorder_10Percentile
        1  : original_firstorder_90Percentile
        2  : plc_status
  ▼ 1  [3]
        0  : -865.767
        1  : -361.656
        2  : 0
```

Figure 3-22 Sample result of scatterplot data returned

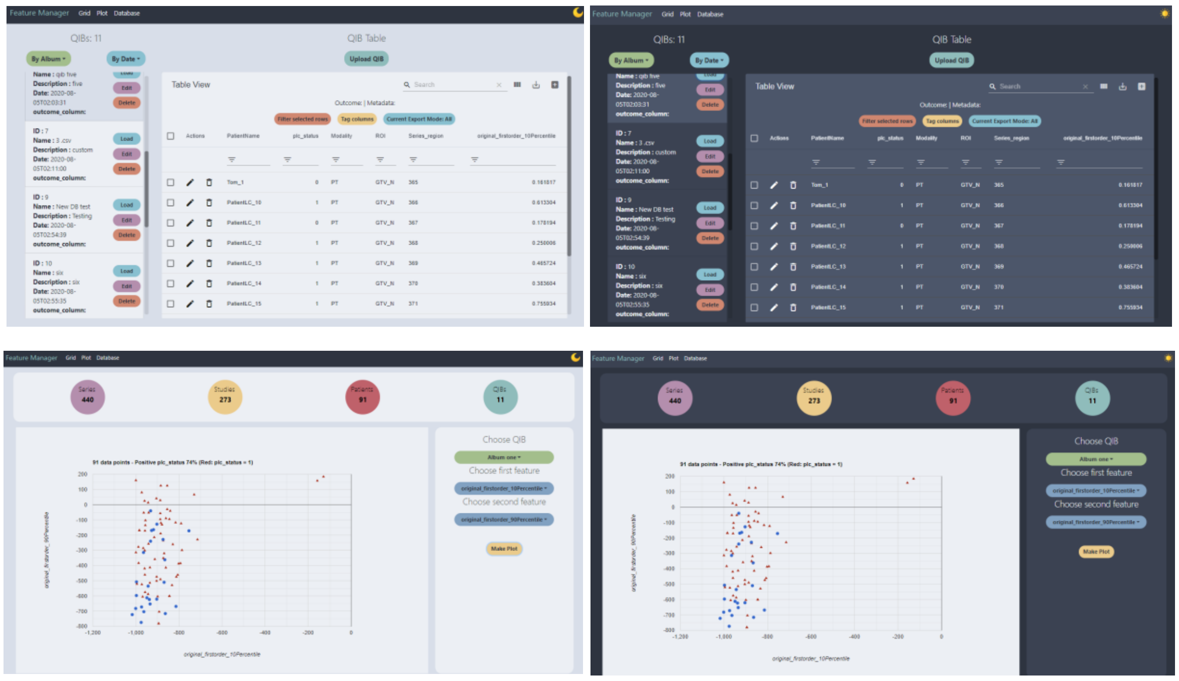
3.3 Front-end

The front-end application is divided into 3 main parts:

- Grid view: users can query for QIBs and load them into an editable Table, as well as upload new CSV files. This is the default part.
- Plot view: users can generate bivariate scatter plots of any two features from a QIB. General statistics of the database are also displayed here.
- Database view: users can perform CRUD (add, edit, delete) operations on various metadata entities.

3.3.1 Dark mode

Users can switch between light and dark mode by clicking the moon/sun icon at top right corner. Theme reference will be saved as local storage and persists the next time the frontend application launches.



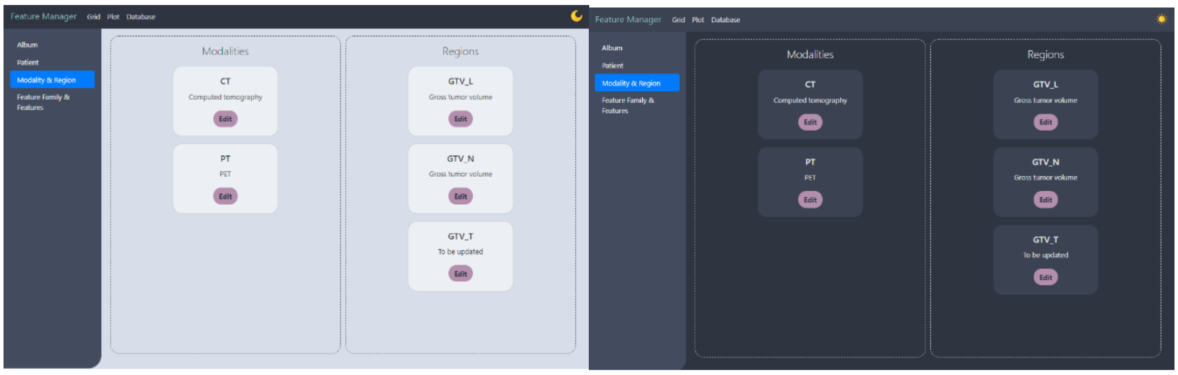


Figure 3-23 Light/Dark mode of Grid, Plot and Database views respectively

3.3.2 Grid View

3.3.2.1 Sorting QIBS

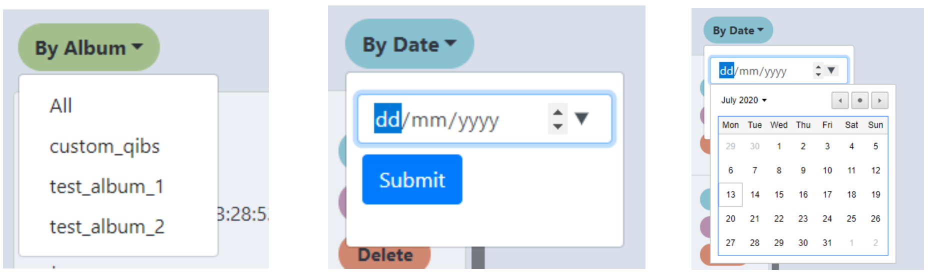
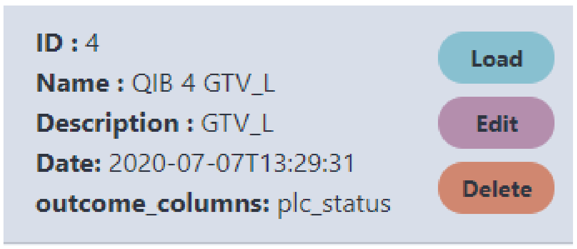


Figure 3-24 Sorting QIB by Albums/Date

QIBs can be sorted by Album or by Date (QIB created since the selected Date). The date selector comes with a calendar for easy selection.

3.3.2.2 Managing QIBs

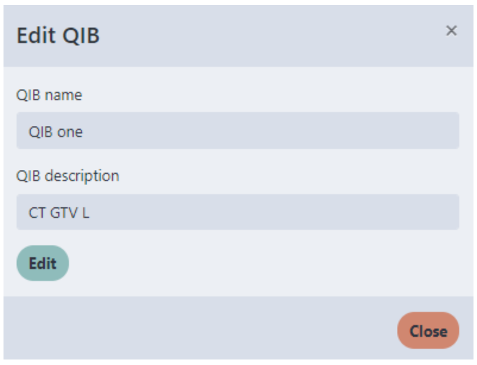


ID : 4
Name : QIB 4 GTV_L
Description : GTV_L
Date: 2020-07-07T13:29:31
outcome_columns: plc_status

Figure 3-25A QIB card menu

For each QIB, users have the option to either load it into table view, edit name and description, or delete the QIB. Loaded QIB card will have a different colour then the others in list.

Clicking the edit button brings up a form for users to type in a name and description of the QIB. Both fields are required. The QIB list will reload after submitting.


Edit QIB ×

QIB name

QIB one

QIB description

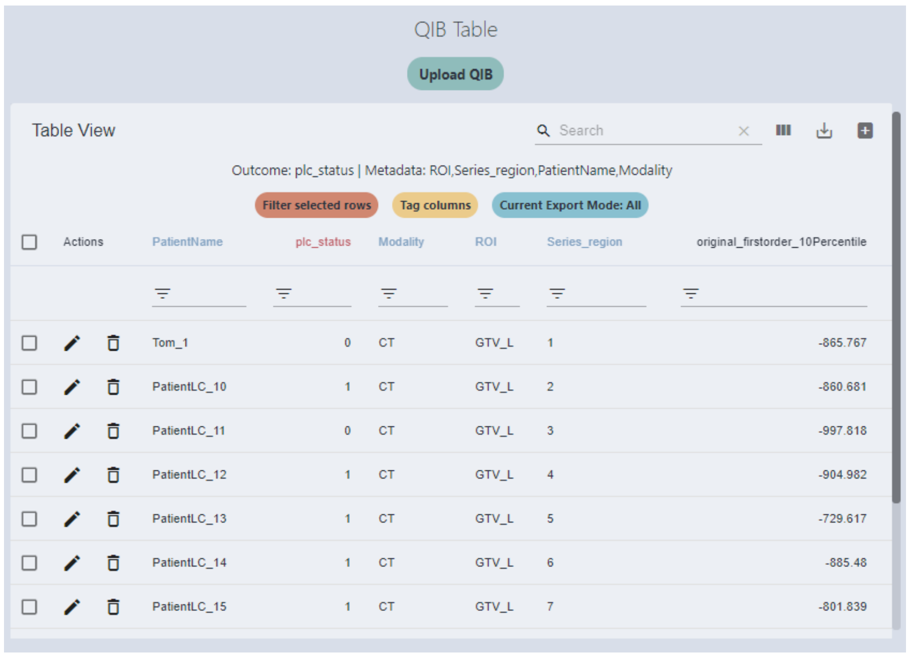
CT GTV L

Edit

Close

Figure 3-26 Edit form

3.3.2.3 QIB Table view


QIB Table

Upload QIB

Table View — Search

Outcome: plc_status | Metadata: ROI,Series_region,PatientName,Modality

Filter selected rows | Tag columns | Current Export Mode: All

☐	Actions	PatientName	plc_status	Modality	ROI	Series_region	original_firstorder_10Percentile
☐	✎ 🗑	Tom_1	0	CT	GTV_L	1	-865.767
☐	✎ 🗑	PatientLC_10	1	CT	GTV_L	2	-860.681
☐	✎ 🗑	PatientLC_11	0	CT	GTV_L	3	-997.818
☐	✎ 🗑	PatientLC_12	1	CT	GTV_L	4	-904.982
☐	✎ 🗑	PatientLC_13	1	CT	GTV_L	5	-729.617
☐	✎ 🗑	PatientLC_14	1	CT	GTV_L	6	-885.48
☐	✎ 🗑	PatientLC_15	1	CT	GTV_L	7	-801.839

Figure 3-27 A loaded table

Sorting

Rows can be ordered by a specific column's value by clicking on the black arrow beside each column's name. Hover mouse over column's name to make the sorting arrow appear.

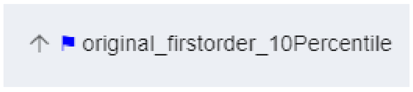


Figure 3-28 Sorting arrow

Filtering

Rows can be filtered by typing in the filter input for column search, or the search bar for a wider search.

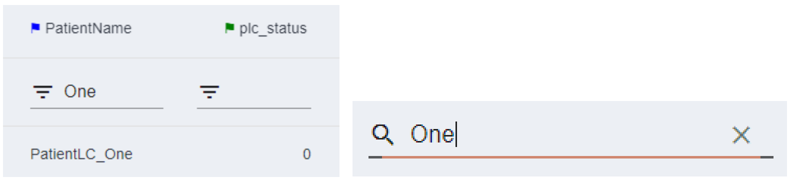


Figure 3-29 Filtering for patient name

Pagination

Users can select the number of rows appear in each page (5-10-20 rows), and navigate pages using the arrows.

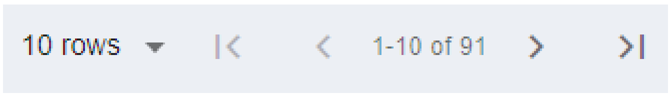


Figure 3-30 Pagination control at the bottom of table view

Editing

Users can edit, delete and add rows by clicking on the following buttons, but changes are only static, and can be exported to CSV. They are **not** persisted in the database.

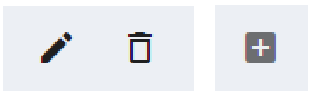


Figure 3-31 Edit, delete, add rows

Column tagging

Users can tag columns as outcome or as metadata column by clicking on Tag columns. Tagged columns shows up as red for outcome and blue for metadata in the table. Column tags will be also set as file's name on export.

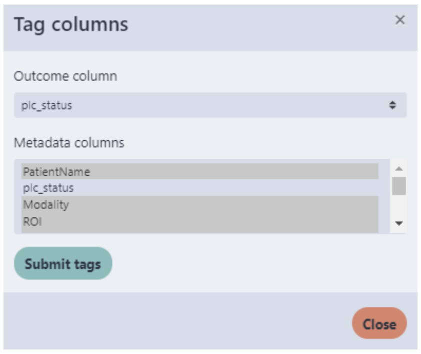


Figure 3-32 Column tagging form

Exporting

To export the table into CSV, users can click on button.

Users can either export only the current page or all rows by toggling between Current Export Mode: All and Current Export Mode: Current Page.

To select columns for export, users can click on the and toggle columns they want to export.

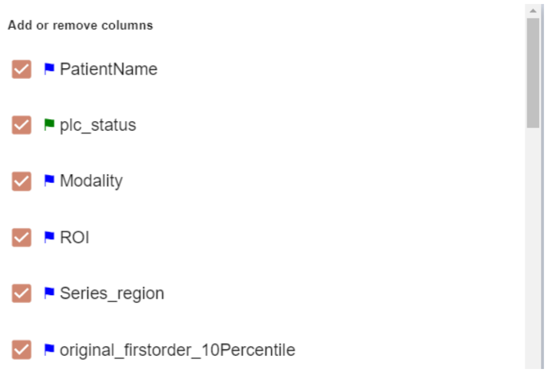
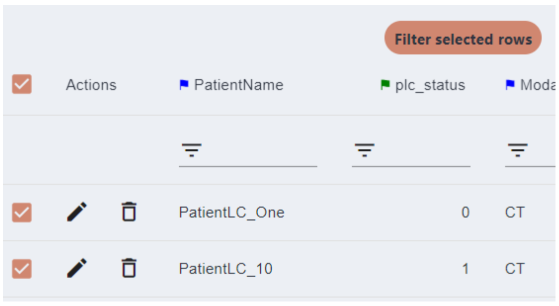


Figure 3-33 Toggling columns

To select rows for export, check the checkboxes next to rows for export, and click Filter selected rows. To undo rows selection, reload the QIB.



	Actions	PatientName	plc_status	Moda
✓		PatientLC_One	0	CT
✓		PatientLC_10	1	CT

Figure 3-34 Toggling rows

3.3.2.4 Uploading QIBs

To open the upload menu, click on Upload QIB

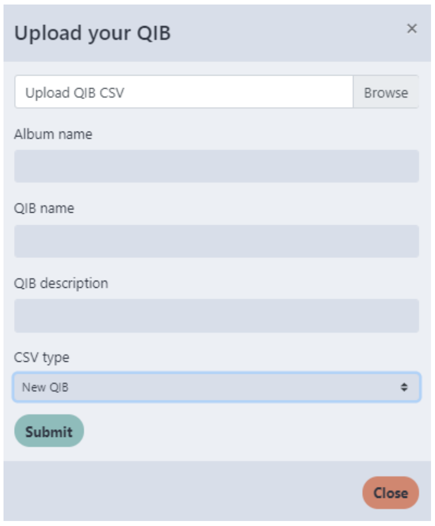
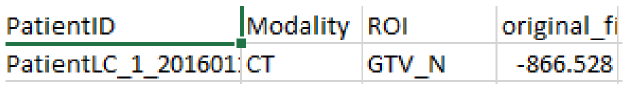


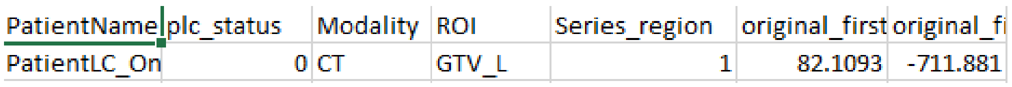
Figure 3-35 Upload form

All fields are required. However depending on the QIB type, the CSV file will look different. Only the feature columns can be added or removed, but others are required.



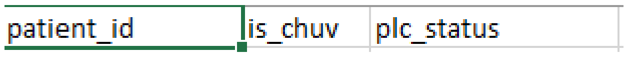
PatientID	Modality	ROI	original_fi
PatientLC_1_201601:	CT	GTV_N	-866.528

Figure 3-36 Valid new QIB file



PatientName	plc_status	Modality	ROI	Series_region	original_first	original_fi
PatientLC_On	0	CT	GTV_L	1	82.1093	-711.881

Figure 3-37 Valid custom QIB file



patient_id	is_chuv	plc_status

Figure 3-38 Valid outcome list file

3.3.3 Plot View

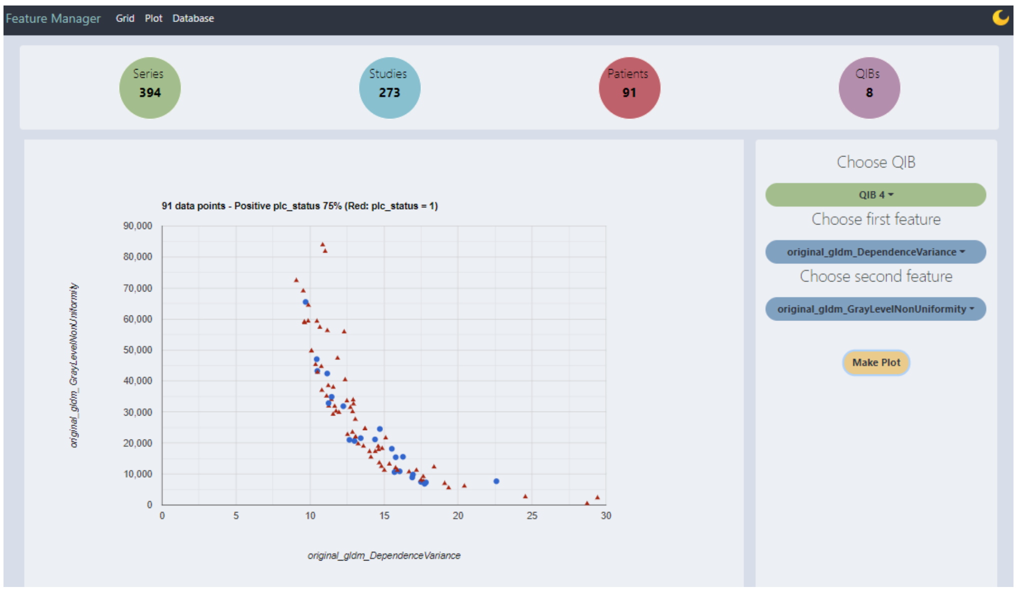

Feature Manager
Grid
Plot
Database
Series
394
Studies
273
Patients
91
QIBs
8
91 data points - Positive plc_status 75% (Red: plc_status = 1)
original_gldm_GrayLevelNonUniformity
original_gldm_DependenceVariance
90,000
80,000
70,000
60,000
50,000
40,000
30,000
20,000
10,000
0
0
5
10
15
20
25
30
Choose QIB
QIB 4
Choose first feature
original_gldm_DependenceVariance
Choose second feature
original_gldm_GrayLevelNonUniformity
Make Plot

Figure 3-39 Plot view

3.3.3.1 Statistics

The bar with 4 circles depicts current number series, studies, patients, and QIBs in the database.

3.3.3.2 Generating a bivariate scatterplot

To start, users need to click on **Choose QIB**, and choose the QIB they want to visualize. After selection, the list of features present in that QIB will be loaded, and two features can be picked.

To make the plot, users can then click on **Make Plot**.

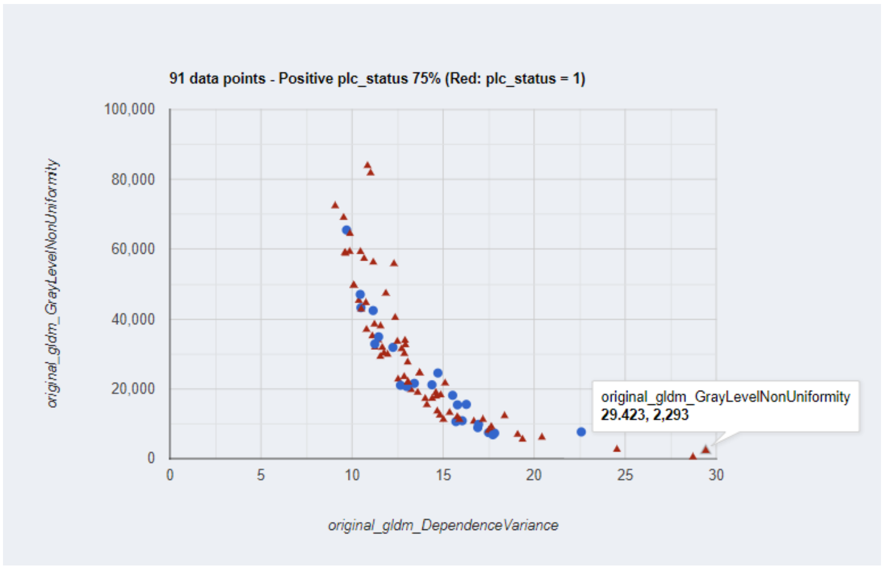


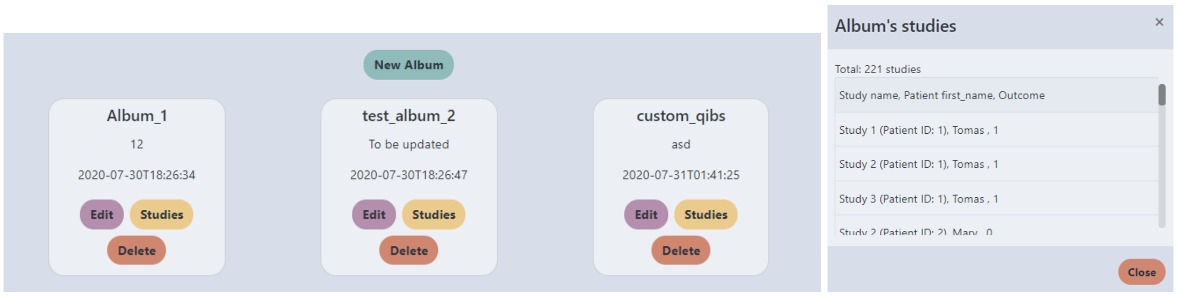
Figure 3-40 Generated scatterplot

Data point information is displayed on mouse hover.

3.3.4 Database View

Database View is divided into 4 tabs on the left sidebar: album, patient, modality/region, and feature/feature family.

3.3.4.1 Album


New Album

Album_1
12
2020-07-30T18:26:34

test_album_2
To be updated
2020-07-30T18:26:47

custom_qibs
asd
2020-07-31T01:41:25

Album's studies

Total: 221 studies

Study name, Patient first_name, Outcome
Study 1 (Patient ID: 1), Tomas , 1
Study 2 (Patient ID: 1), Tomas , 1
Study 3 (Patient ID: 1), Tomas , 1
Study 2 (Patient ID: 2), Mary , 0

Figure 3-41 Album tab and List of selected album's studies

From the album tab users can edit name and description of albums, add new albums, and delete albums. Only albums that have no referenced QIBs can be deleted.

Users can also view the list of studies of each album by clicking **Studies**.

3.3.4.2 Patient

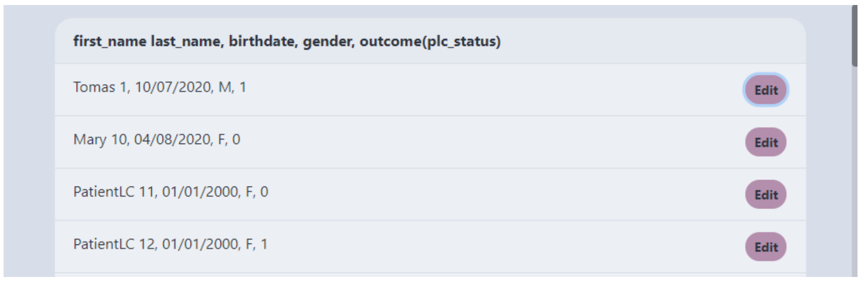
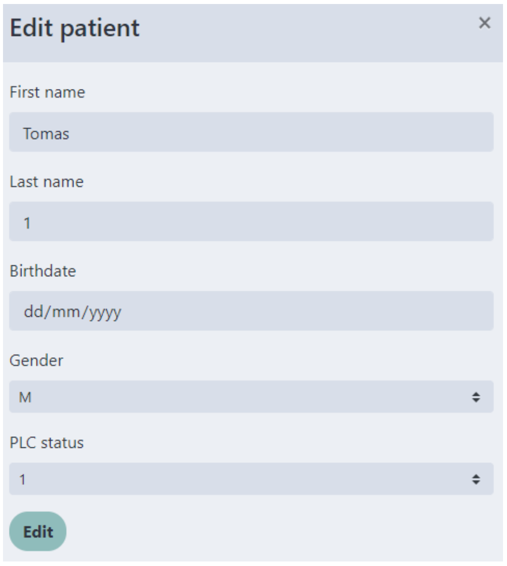

first_name last_name, birthdate, gender, outcome(plc_status)
Tomas 1, 10/07/2020, M, 1
Mary 10, 04/08/2020, F, 0
PatientLC 11, 01/01/2000, F, 0
PatientLC 12, 01/01/2000, F, 1

Figure 3-42 Patient tab

From the patient tab users can view and edit patient information. Editable information includes first_name, birthdate, gender, and plc_status. Last_name is a read-only field and not editable due to the reason mentioned in section 3.2.2.1.

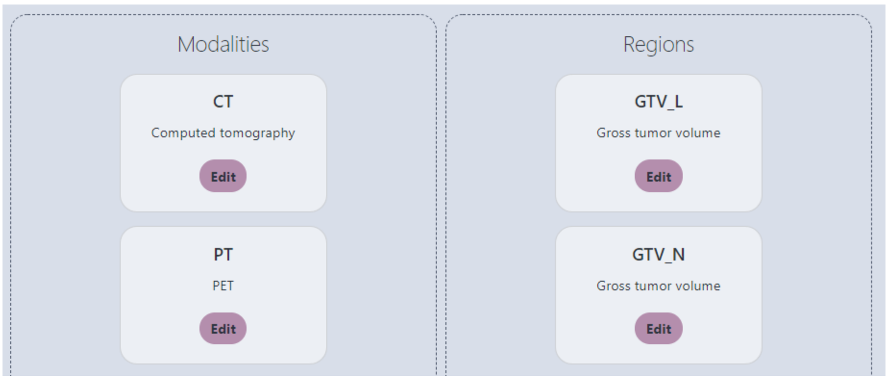


Edit patient

First name	Tomas
Last name	1
Birthdate	dd/mm/yyyy
Gender	M
PLC status	1

Figure 3-43 Patient edit form (Last name is noneditable)

3.3.4.3 Modality & Region



Modalities

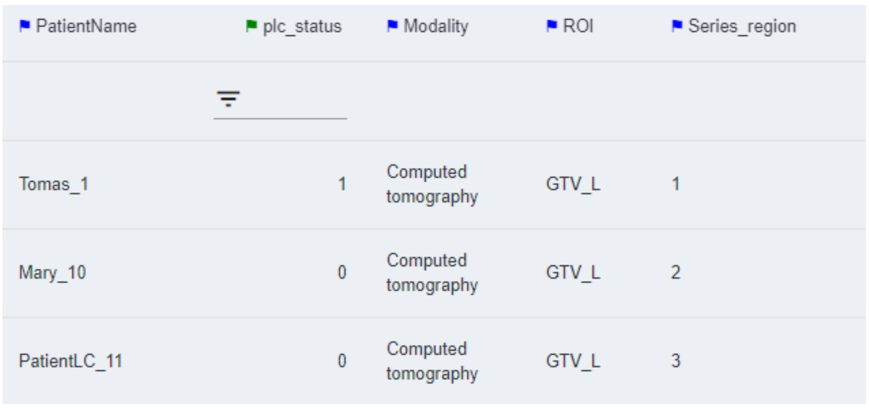
- CT — Computed tomography
- PT — PET

Regions

- GTV_L — Gross tumor volume
- GTV_N — Gross tumor volume

Figure 3-44 Modality & region tab

To preserve data integrity of the database, deleting Modalities/ Regions is restricted, however, they are fully editable, and any changes made here will show up in the loaded Table in Grid View.



PatientName	plc_status	Modality	ROI	Series_region
Tomas_1	1	Computed tomography	GTV_L	1
Mary_10	0	Computed tomography	GTV_L	2
PatientLC_11	0	Computed tomography	GTV_L	3

Figure 3-45 Modality CT's change into 'Computed tomography' is reflected in the table

3.3.4.4 Feature & Feature Family

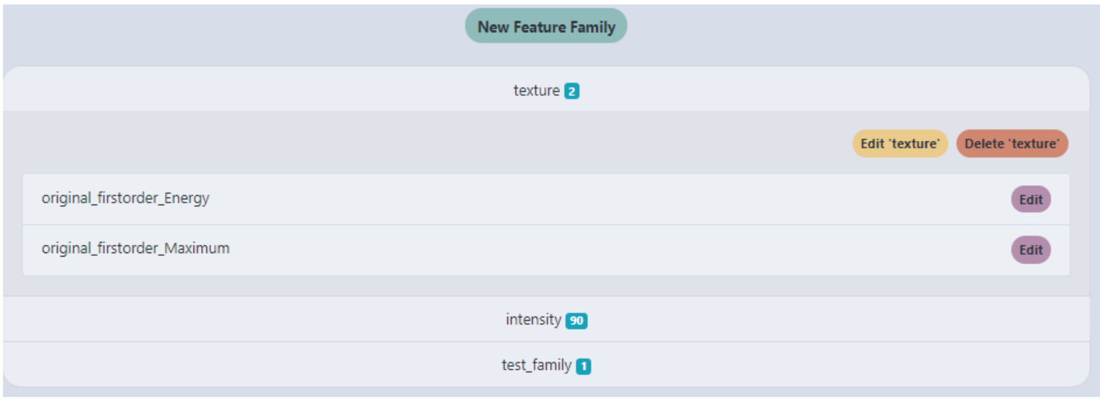


Figure 3-46 Feature & Feature Family tab

From this tab users can

- Add/edit/delete feature families
- Change feature's name and switch features from one family to another

Only families with no features can be deleted. The small blue badge next to each family's name represents how many features there are in that family.

4. DISCUSSION

4.1 Goal evaluation

From section 1.3, this section will discuss how the implemented solution performs compared to the original goals:

- Store extracted feature sets
- Provide an interface to query and interactively compile feature sets in real time.
- Edit metadata of feature sets without compromising data integrity
- Implement simple visualization of data

4.1.1 Storing extracted feature sets

As long as the raw CSV file is in the correct format, then this application will be able to store it into the database. By essentially querying for data according to the initial insertion order explained in section 3.1.3, stored feature sets can be correctly loaded back into table form, while still maintaining advantages of a relational data model.

In addition to raw feature sets, this application can also be used to store custom feature sets (only feature sets filtered using the provided Table in the application) by exporting and reuploading them as a Custom QIB. This allows for saving custom filtered feature sets in the database, since having to save them locally would somewhat defeat the point of the project.

4.1.2 Providing interface to query and interactively compile feature sets in real time

The Grid View (section 3.3.2) part covers this goal. QIBs can be filtered by album and upload date. The QIB Table is full-featured and provides functionalities for columns and rows search/filter/sort/select, as well as CSV export. Users can directly search for a QIB, load it, customize it and export it into CSV from the web interface without looking at any raw CSV feature sets.

4.1.3 Editing metadata of feature sets without compromising data integrity

CRUD operations can be performed on some of the entities from the web application interface. The symbol * means the operation is limited to an extent. The completeness of CRUD operations implemented will be measured by **CRUD Availability rate.**

While not all entities should be weighted the same (Series and Studies have nearly no impact on the application's performance, for example), this will give an approximation of how much the database is available for use from the web application interface.

Method of calculation:

For each entity:

- An available CRUD operation counts as 25%
- An available CRUD operation with limitations (* symbol) counts as 12.5%
- Maximum rate is 100% for all four available CRUD operations with no limitations.

Total Availability rate is the average rate of all entities.

A.Rate (%)	Entity	CRUD operations available	Reason for limitations
87.5	Album	Create, Read, Update, Delete*	Deleting albums with children QIBs with set those QIB's album to null, which makes them unsearchable unless all albums are displayed.
25	Study	Read	Studies info are filled with mock data, they currently do not affect the app's functionalities
0	Series	None	Series info are filled with mock data, they currently do not affect the app's functionalities
50	Modality	Read, Update	Deleting modalities will irreversibly affect all related entities (series, series_region, qib_feature)
50	Region	Read, Update	Deleting regions will irreversibly affect all related entities (series_region, qib_feature)
87.5	QIB	Read, Create*, Update, Delete	Uploading works if the input CSV has correct columns.
25	QIBFeature	Read	Entity is too granular and connected for CUD operations. Can be edited in QIB Table, but changes will not be saved to database.
50	Feature	Read, Update	Deleting feature will irreversibly affect related QIBs.
100	Family	Read, Create, Update, Delete	Can only delete feature families with no feature. Switch all children features to another family before deletion.
37.5	Patient	Read, Update*	Cannot update last_name (section 3.2.2.1)
50	Outcome	Read, Update	No point in creating orphan outcomes with no patient.
51.1%			**Average CRUD availability rate**

Figure 4-1 CRUD Availability Table

4.1.4 Visualization of feature sets

The application provides some visualization functionalities, but it was not the main focus of this project. The current front-end application implements a visualization in the form of bivariate scatter plots, and the relational data model is capable of supporting other types of visualization. An example of another type visualization would be outcome distribution by patient's gender.

4.2 Advantages of a modular radiomics solution

"Modular solution", in this context, means solutions that are not full-package and support all four steps in the radiomics workflow (Load, Extract, Storage, Machine Learning) (section 1.2.3).

This thesis project argues for a more modular approach to implementing radiomics research projects.

Flexibility: Full-package solutions, such as i2b2, can suffer from long term bugs and downsides that are, due to its scale, hard to address. By keeping components of the solutions modular and interchangeable, it is easier to isolate a problem in each component, or to change them out completely.

Case in point, this project's REST API can be reused with any front-end client. From the backend side, the data model can be modified without breaking the front-end client as long as then REST endpoints remain the same.

Scalability: It is not efficient to implement the full-package solution for small scale projects. Modular solutions can scale to accommodate more types of projects.

Specialization: Modular solutions allow for implementing only components relevant to the goal of the project, without having to install unnecessary components.

Accessibility and Testing: Modular solutions are easier to set up and carry out unit tests. One of i2b2's drawbacks is its difficulty in installation and testing (Wagholikar KB, 2018). This project application was developed with Python and JavaScript, which are popular programming languages with well-developed testing frameworks.

4.3 Values added

This project shows that storing feature sets using relational entity model is feasible and desirable. There are still points of improvement with the data model (to be discussed in section 4.4), but in general, it can be implemented well.

The project also shows benefits of managing feature sets using the relational entity model. Compared to manually sorting through CSV files, querying and editing metadata entities in the feature manager is far more efficient. It is possible to both view feature sets in their original table format and enjoy the perks of a relational data model (e.g. metadata query and visualization).

Additionally, utilization of Object-Relational mapping (SQLAlchemy) in the back-end application has been very successful and is highly recommended for future projects. However, in some cases there can be performance differences between ORM calls and regular SQL calls, so the documentation should be checked carefully.

4.4 Points of improvement

This section will discuss the current drawbacks of the application and room for future improvement.

4.4.1 Data model

The data model is rigid and heavily dependent on the column structure and quirks of input CSV files. This has been a major hurdle in the course of developing this project. Changes in the CSV file metadata structure can result in total change in the data model. For example, the Outcome table relies entirely on the CSV file having the PatientID field in correct format (section 3.2.2.1).

As such, it is not possible to upload any random feature set into the application without checking if the columns and data format are correct.

A possible solution would be to implement a live csv editor in the application front end before submitting the file. However, this would only be of use if the user does not have access to an existing CSV editor (Excel for example). In any case, a stable and unchanging CSV file standard is needed in order to make this data model work.

Some of the entities in the data model are very interconnected with each other and it is not possible to delete or modify them without significantly affecting the others and the overall integrity of the database.

With the current project's scope, only plc_status is used as the outcome for patients, and thus the application can only support feature sets relevant to plc_status. A point of improvement would be to add more outcome properties in the data model's Outcome table, to accommodate different types of dataset. This will also make the Column tagging feature in the front-end web interface much more useful.

4.4.2 Back-end

In a similar fashion, the ETL is also dependent on the format of input CSV files. The ETL process can be slow, especially if connected to a remote database and internet connection is weak.

Another point of improvement was the fact that this project has not managed to combine querying and loading multiple QIBs at once. The current function to convert QIBFeatures into Table dictionary requires the resulting dictionary to have no empty rows and columns, which would be the case if the QIBs contain different feature columns/ different number of rows. A possible solution would be to comb the selected QIB beforehand and taking only intersecting columns before conversion.

4.4.3 Front-end

The QIB table provides cell and column CRUD, but changes are not saved to the database due to integrity issues. Exporting a CSV and immediately reuploading them works, but if the user makes any edit to the local file before reupload, especially to the Series_region column, the saved file might not be stored correctly in the database.

As discussed in section 4.4.1, with an Outcome table with multiple outcome properties, the web interface could implement dynamic outcome column loading, and user can choose which outcome to display in table view or in visualization.

4.4.4 Visualization

Current visualization capability is simple and limited, but there is room for improvement in this area. The current visualization library used is react-google-charts, but there are many other more extensive libraries available. One suggestion is **Plotly**, a JavaScript open source graphing library based on d3.js and stack.gl.

4.4.5 Security

One area this project has not implemented was security. It would be good to implement user login to the application and enable user roles. For example, guests and regular users can only add and manage QIBs and Albums, while superusers can edit/delete other metadata like Modality and Region of Interest.

5. CONCLUSION

This thesis project aims to create a fully usable tool to manage feature sets and enable user to store, filter, customize and export them without opening any CSV file. To that end, the project has mostly achieved the original goal, albeit with some limitations.

One of the more challenging part of developing this application was the design of the relational data model to transform a CSV table into relational model, and still be able to query and reformat them back into table form. This problem has been solved at the cost of enforcing a very rigid data model that might not adapt well to changes in CSV input.

The dataset used for populating this project’s database was limited (less than 10 CSV files). As such, this project application has only been confirmed to work in testing, and much further improvement and testing need to be done before it can be used in production.

To start on improving this application, it is recommended to pull this project from GitHub and try it out first, as the data model and data sets can be populated with a few commands. The focus point for improvement of this project is on the database model to make it more flexible and forgiving to new changes, along various other possible improvements.

Overall, this thesis project has been a good challenge in data modelling and Python/JavaScript programming and an invaluable learning experience. Despite various drawbacks, the project is fairly successful and hopefully it will be of help to future radiomics project.

REFERENCES

Dias, C., 2015. *VSCode Program Manager's comment on VSCode licenses situation.* [Online]
Available at: https://github.com/Microsoft/vscode/issues/60#issuecomment-161792005
[Accessed 29 6 2020].

Documentation, S., n.d. *SQLAlchemy Documentation.* [Online]
Available at: https://docs.sqlalchemy.org/en/13/orm/cascades.html#cascade-merge
[Accessed 30 6 2020].

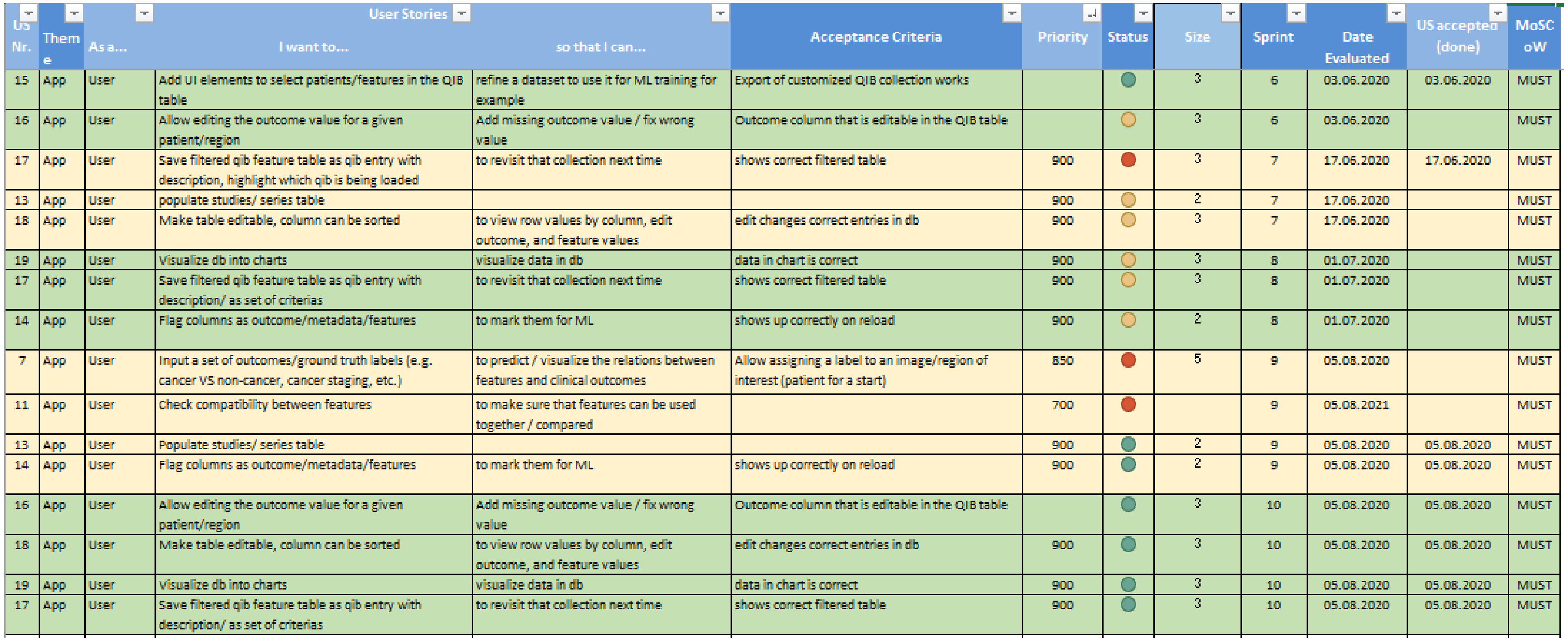
Naim Qaqish, F. G., n.d. *Lymphangitic carcinomatosis.* [Online]
Available at: https://radiopaedia.org/articles/lymphangitic-carcinomatosis
[Accessed 23 7 2020].

Sara Ranjbar, J. R. M., 2017. An Introduction to Radiomics: An Evolving Cornerstone of Precision Medicine. In: *Biomedical Texture Analysis: Fundamentals, Tools and Challenges.* London: Elsevier Ltd., p. 223.

Wagholikar KB, M. M. D. P. e. a., 2018. Automating Installation of the Integrating Biology and the Bedside (i2b2) Platform. *Biomed Inform Insights,* pp. 1-2.

APPENDIX I PRODUCT BACKLOG

US Nr.	Theme	As a...	User Stories I want to...	so that I can...	Acceptance Criteria	Priority	Status	Size	Sprint	Date Evaluated	US accepted (done)	MoSCoW
0	Preparation	Developer	Setup my working environment (Teams, PyCharm, Git repo, etc.)	I will be ready to work on the US in sprint 1	GitHub Repo Set up, tools installed, Teams set up	1000	🟢	1	0	04.03.2020	04.03.2020	MUST
1	Preparation	Developer	Look at existing technologies for storing features, collections & outcomes	to find the optimal solution to store all the data	Knowledge of Python & libraries shown	990	🟢	3	1	04.03.2020	04.03.2020	MUST
2	App	User	Create an entity diagram for storing the features	to be able to store, retrieve/query the data efficiently	Show the first version of the diagram	995	🟢	5	1	25.03.2020	25.03.2020	MUST
3	App	Developer	Define an object model for storing the features	to be able to store, retrieve/query the data efficiently	Show object model	980	🔴	5	1	25.03.2020		MUST
4	App	User	Display of a welcome screen	to navigate in the various functionalities proposed	Show a skeleton of an app with a menu	970	🟢	1	1	25.03.2020	25.03.2020	MUST
5	App	Developer	Make a comparsion between pandas, i2b2, etc.	to demonstrate an objective reason for the technological choice	Show us the comparison table	950	🟢	2	1	25.03.2020	25.03.2020	MUST
6	App	User	See an overview of my feature collections	to be able to view/filter/select features to analyze	Show us a page listing features, filter by album name	900	🔴	5	2	08.04.2020		MUST
5	App	Developer	In-depth analysis and comparison between i2b2 (as a platform) VS SQLAlchemy+Pandas	to better understand the differences, limits of i2b2, etc.	Show us the updated table	860	🔴	2	2	08.04.2020		MUST
7	App	User	Input a set of outcomes/ground truth labels (e.g. cancer VS non-cancer, cancer staging, etc.)	to predict / visualize the relations between features and clinical outcomes	Allow assigning a label to an image/region of interest (patient for a start)	850	🟡	5	2	08.04.2020		MUST
8	App	User	Use pandas to load csv into model	to load csv files into mysql db and display them	Information from csv file shows up properly in database	900	🟢	3	3	22.04.2020	22.04.2020	MUST
6	App	User	See an overview of my feature collections in the React front end	See and sort features by album	Features display correctly by criteria	900	🟢	4	3	22.04.2020	22.04.2020	MUST
5	App	User	In-depth analysis and comparison between i2b2 (as a platform) VS SQLAlchemy+Pandas, why chose Flask	to better understand the differences, limits of i2b2, etc.	Show us the updated table (Note: look more into how i2b2 integrate with ML (esp. python frameworks))		🟢	2	4	06.05.2020	06.05.2020	MUST
9	App	User	Design use cases	to better understand how to make the app UI	Use case chart	900	🟢	2	4	06.05.2020	06.05.2020	MUST
5	App	User	See an overview of my feature collections in the React front end	See and sort features by date	Features display correctly by criteria	900	🟢	1	4	06.05.2020	06.05.2020	MUST
10	App	User	Edit/delete/filter/select features	to build a finalized feature set for ML	export feature set into CSV , simple ui	900	🟢	4	4	22.05.2020	22.05.2020	MUST
11	App	User	Check compatibility between features	to make sure that features can be used together / compared		700	🟡		5	22.05.2020		MUST
12	App	User	Display table of QIB values	to see the QIB values and choose features for feature set	see the table display correct values	900	🟢	3	5	22.05.2020	22.05.2020	MUST
13	App	User	Generate dummy values for metadata entities	to be able to query modality/region_of_interest/outcome from metadata	query returns correct values	900	🟢	4	5	22.05.2020	22.05.2020	MUST
14	App	User	Generate feature set with ground truths labels	to be able to use feature set for ML	query returns correct values	900	🟡	3	5	22.05.2020		MUST



US Nr.	Theme	As a…	User Stories: I want to…	User Stories: so that I can…	Acceptance Criteria	Priority	Status	Size	Sprint	Date Evaluated	US accepted (done)	MoSCoW
15	App	User	Add UI elements to select patients/features in the QIB table	refine a dataset to use it for ML training for example	Export of customized QIB collection works		🟢	3	6	03.06.2020	03.06.2020	MUST
16	App	User	Allow editing the outcome value for a given patient/region	Add missing outcome value / fix wrong value	Outcome column that is editable in the QIB table		🟠	3	6	03.06.2020		MUST
17	App	User	Save filtered qib feature table as qib entry with description, highlight which qib is being loaded	to revisit that collection next time	shows correct filtered table	900	🔴	3	7	17.06.2020	17.06.2020	MUST
13	App	User	populate studies/ series table			900	🟠	2	7	17.06.2020		MUST
18	App	User	Make table editable, column can be sorted	to view row values by column, edit outcome, and feature values	edit changes correct entries in db	900	🟠	3	7	17.06.2020		MUST
19	App	User	Visualize db into charts	visualize data in db	data in chart is correct	900	🟠	3	8	01.07.2020		MUST
17	App	User	Save filtered qib feature table as qib entry with description/ as set of criterias	to revisit that collection next time	shows correct filtered table	900	🟠	3	8	01.07.2020		MUST
14	App	User	Flag columns as outcome/metadata/features	to mark them for ML	shows up correctly on reload	900	🟠	2	8	01.07.2020		MUST
7	App	User	Input a set of outcomes/ground truth labels (e.g. cancer VS non-cancer, cancer staging, etc.)	to predict / visualize the relations between features and clinical outcomes	Allow assigning a label to an image/region of interest (patient for a start)	850	🔴	5	9	05.08.2020		MUST
11	App	User	Check compatibility between features	to make sure that features can be used together / compared		700	🔴		9	05.08.2021		MUST
13	App	User	Populate studies/ series table			900	🟢	2	9	05.08.2020	05.08.2020	MUST
14	App	User	Flag columns as outcome/metadata/features	to mark them for ML	shows up correctly on reload	900	🟢	2	9	05.08.2020	05.08.2020	MUST
16	App	User	Allow editing the outcome value for a given patient/region	Add missing outcome value / fix wrong value	Outcome column that is editable in the QIB table		🟢	3	10	05.08.2020	05.08.2020	MUST
18	App	User	Make table editable, column can be sorted	to view row values by column, edit outcome, and feature values	edit changes correct entries in db	900	🟢	3	10	05.08.2020	05.08.2020	MUST
19	App	User	Visualize db into charts	visualize data in db	data in chart is correct	900	🟢	3	10	05.08.2020	05.08.2020	MUST
17	App	User	Save filtered qib feature table as qib entry with description/ as set of criterias	to revisit that collection next time	shows correct filtered table	900	🟢	3	10	05.08.2020	05.08.2020	MUST

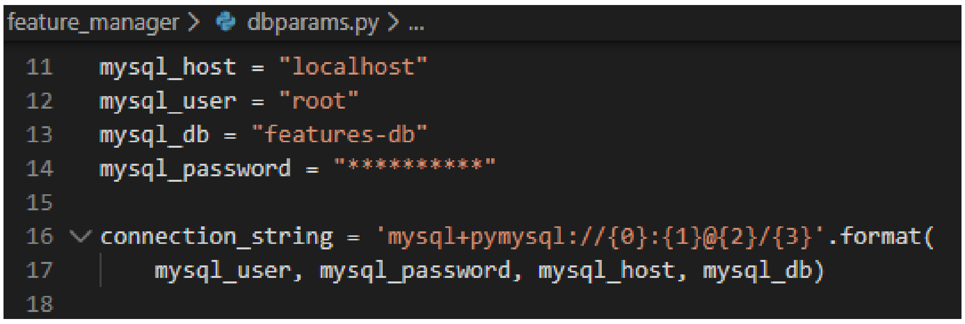
APPENDIX II TECHNICAL GUIDE

GitHub repositories link:

- Back-end: https://github.com/genttunn/python-rest-api.git
- Front-end: https://github.com/genttunn/feature-manager

To set up the project:

- Setup a MySQL server, either online or local. Create a database called 'features-db'
- Install node.js
- Pull this project's back-end and front-end from Github.
- Open the back-end application (python-rest-api), install the requisite package with pip.
- Make a file in python-rest-api/feature_manager/ called dbparams.py, fill it like picture with the variables being the created MySQL from above:



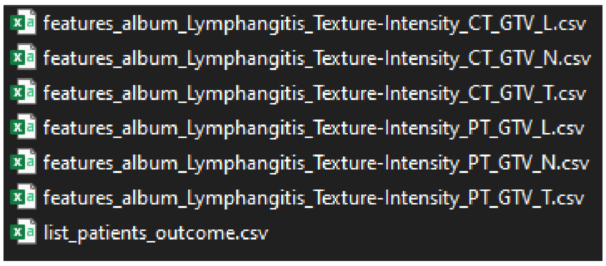
feature_manager > dbparams.py > ...

```python
mysql_host = "localhost"
mysql_user = "root"
mysql_db = "features-db"
mysql_password = "**********"

connection_string = 'mysql+pymysql://{0}:{1}@{2}/{3}'.format(
    mysql_user, mysql_password, mysql_host, mysql_db)

```

- Open a terminal at python-rest-api/ and type *python* to open a Python console there
- Type the following commands one by one to create the data model in database and insert some basic metadata:

```python
from feature_manager import db
db.create_all()
from feature_manager.routes import *
quick_start()
```

- Finally, quit Python console and start the back-end application with *python app.py*
- Open a terminal in the front-end application (feature-manager), type *npm install* to install dependencies, then *npm start* to start the React application. The app is now empty because there are no QIB uploaded yet.
- From the web app, upload the following given CSV files to the database. These files can be found in python-rest-api/csv/:



features_album_Lymphangitis_Texture-Intensity_CT_GTV_L.csv
features_album_Lymphangitis_Texture-Intensity_CT_GTV_N.csv
features_album_Lymphangitis_Texture-Intensity_CT_GTV_T.csv
features_album_Lymphangitis_Texture-Intensity_PT_GTV_L.csv
features_album_Lymphangitis_Texture-Intensity_PT_GTV_N.csv
features_album_Lymphangitis_Texture-Intensity_PT_GTV_T.csv
list_patients_outcome.csv