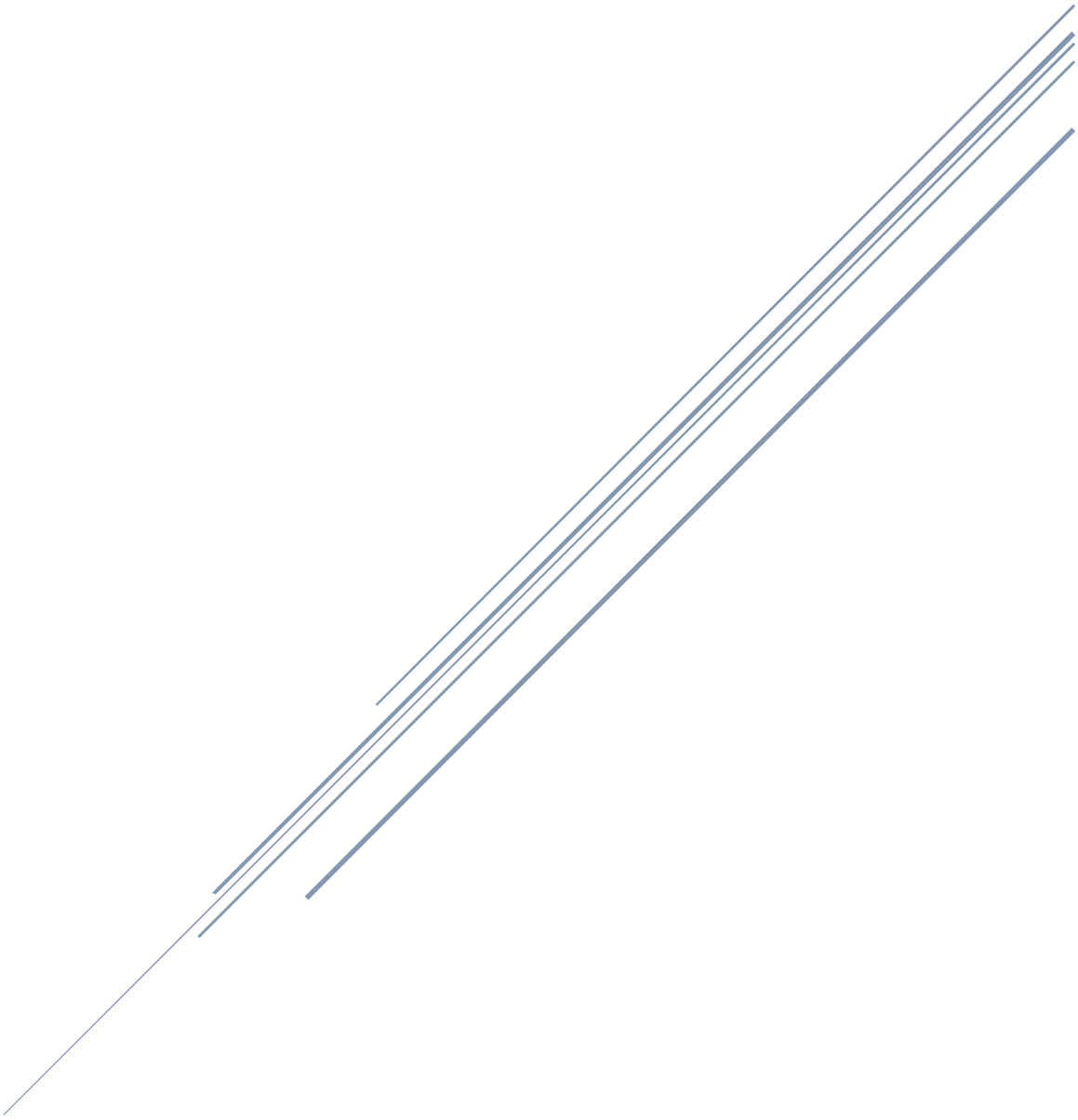
OPENVESSEL DRILL PROJECT OUTLINE



OpenVessel Drill Project Outline

Table of Contents

Overview

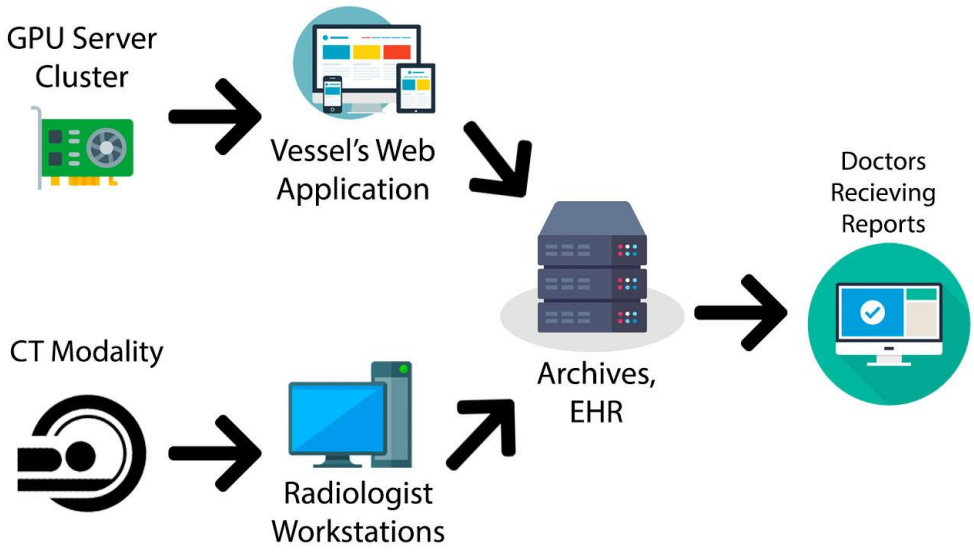
OpenVessel's Drill

OpenVessel's Drill is a downloadable server software for healthcare servers to display internal web applications to users. To upload medical data, transform it with machine learning tools and share across a network. Vessel's Drill integrates following the HL7 protocol, and DICOM protocols. All DICOM files are anonymized from various Modalities and distribute it via doctors to workstations.

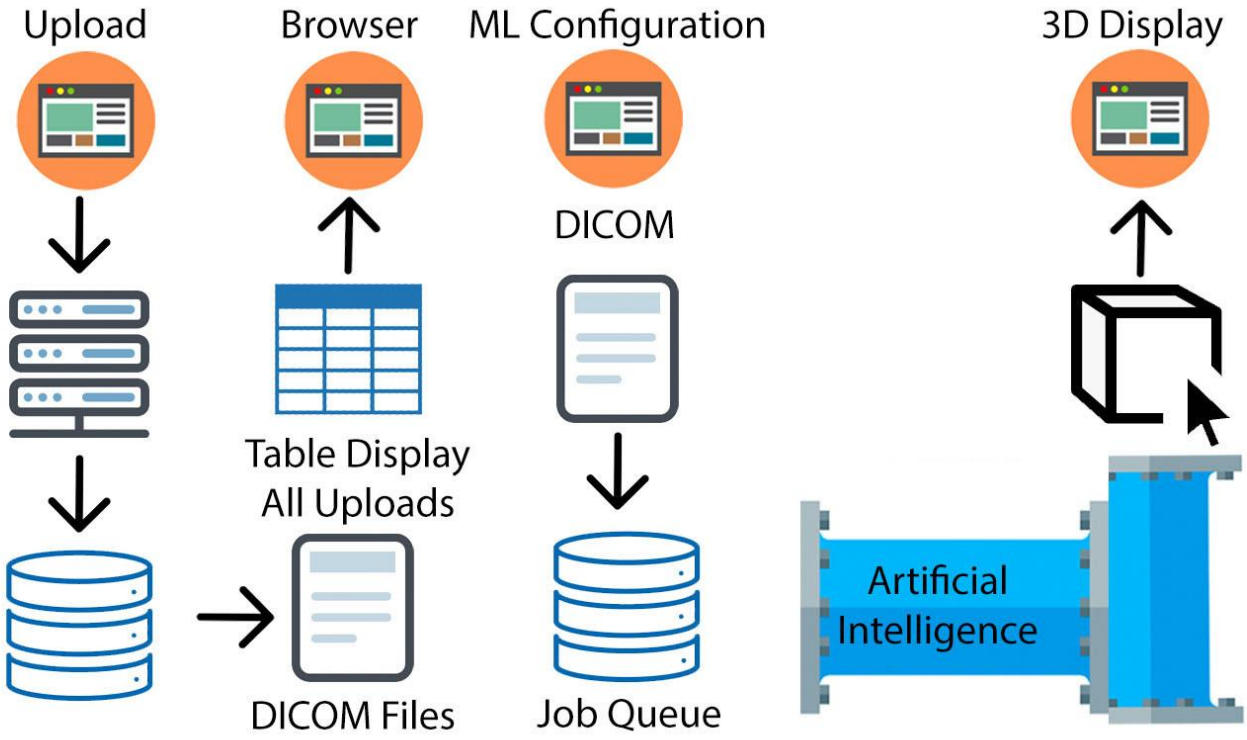
We call it the "Drill" because the server software is a generalized tool to be pointed in any direction to satisfy any particular purpose. The "bits" of the drill, however, are specific. The machine learning tools are analogous to the bits and are case specific and interchangeable; the same as the drill bits themselves.

Vessel's Drill consists of two parts:

1. A web application that host the user web pages
2. Its sister server - the "Data Pipeline", or the "bits" of the Drill - is deployed on a GPU cluster or associated with an external cloud of the medical information system or created internally.



Workflow of the Web Applications



The Web Application

The web application that runs on the GPU clusters communicates with its sister server which is where all the computational heavily lifting is done. This section of this document aims to explain how the web app works and how it communicates with the server hosting the machine learning models. This section is split into 2 sections, being:

1. Front-end (HTML + CSS)
2. Back-end (Flask app + JavaScript)

Front-end

Our web application (the front-end) is how users interact with our product; How it looks, feels, and works. Functionally is essential for us to execute correctly and efficiently. OpenVessel's software is built upon the Flask app web framework that wraps around Werkzeug and Jinja2.

Werkzeug handles all the functions of a WSGI. WSGI is a Web Server Gateway Interface for Python. In simple terms it is how a web server talks to a web application to process a web request. The web server is what manages **requests** from the internet so when a client navigates to our webpage it sends a **request** to our web server. The web server receives the request, processes it, and sends back a **webpage** that the client receives.

Overview of pages in the webapp

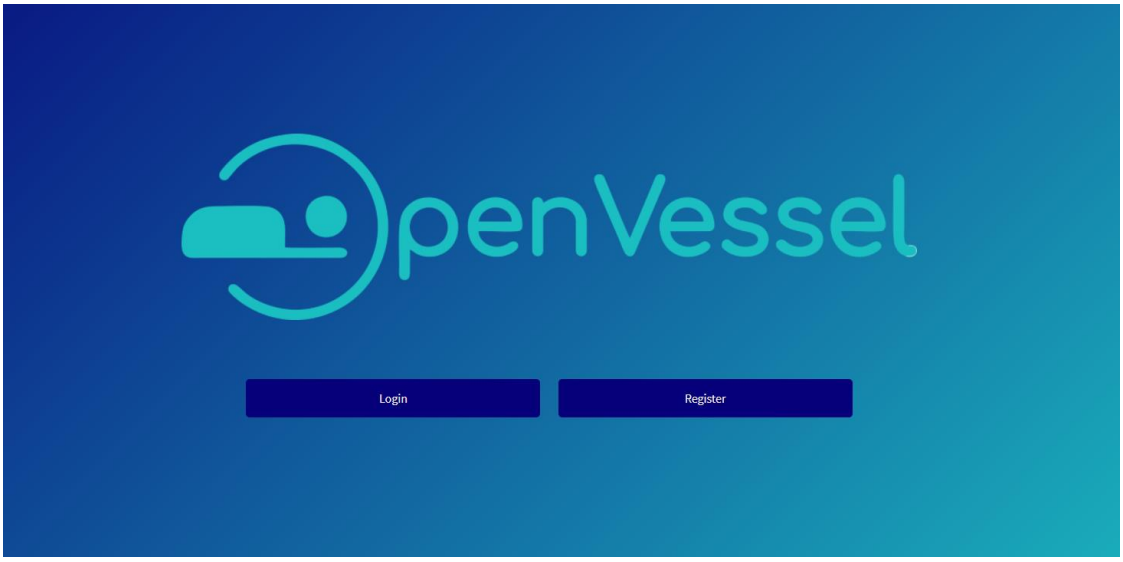
The Front-end consists of these webpages:

- **Home webpage:**
- **Login/Register webpages**: Login, Registration, and User Authentication
- **Account page**: view, edit, and delete account
- **Upload webpage**: Drag and Drop boxes to upload sets of DICOM files
- **Browser webpage**: Browser for viewing uploaded data stored in the database in a paginated format.
- **Job Submission webpage**: Allows the user to send DICOM files to the Data Pipeline
- **3D Viewer webpage**: Displays the result of the Data Pipeline in a 3D VTK displayer embedded in the webpage.

All these webpages are written in HTML and JavaScript and use CSS to improve their aesthetics.

Home Page

The welcome page of the website, has login and register.



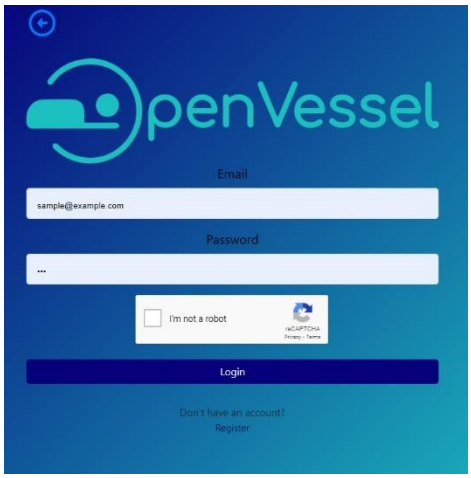
Login/register pages

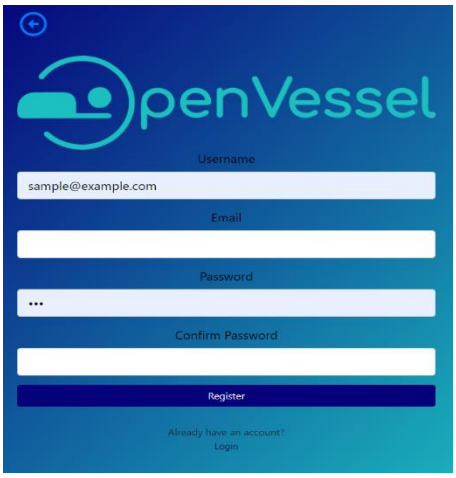
These two webpages handle user authentication. Users can reset their password with a standard “forgot your password” functionality that emails them a password reset link that expires after a period of time.

All the code is found in **auth** folder. Contains forms.py and routes.py

Contains classes

- RegistrationForm() Validates the user’s inputs
- LoginForm() Validates the user’s inputs
- UpdateAccountForm() on the account page

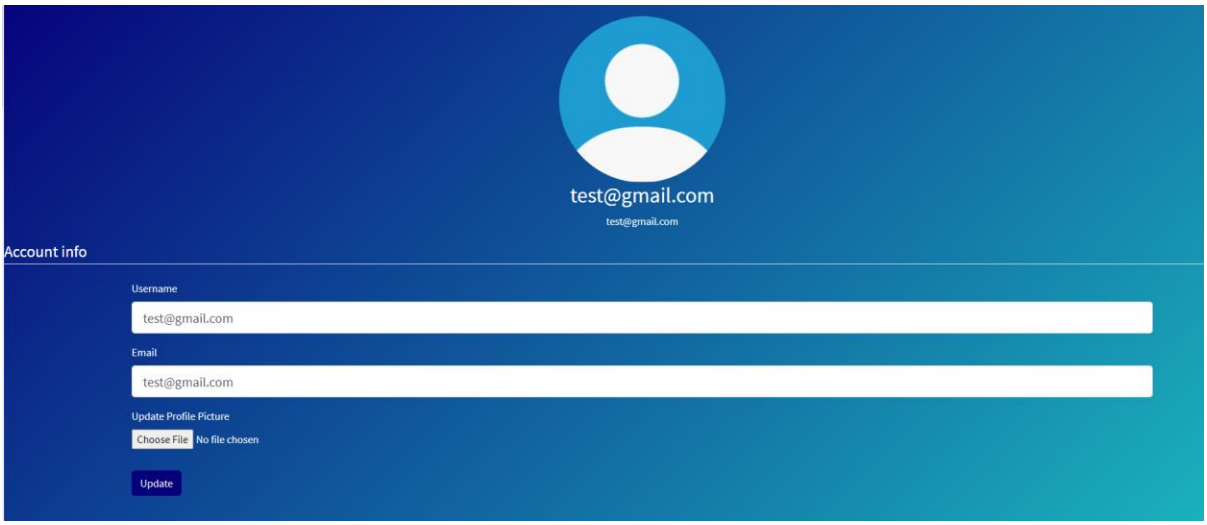




Password storage: For the safety and security of the user and the data that they upload to the database, the password is not directly stored in the database, but rather stored as hashed values using a Flask extension called Flask-Bcrypt.

Account page

The account code is found in **auth** folder. Contains forms.py. The user can view, edit, and delete their account on this page.



The new inputs made on the account page is validate on the submission of data. If a new photo is chosen the image is converted to bytes and saved to the database. The web page should flash “Your account has been updated!” if the upload was successful.

Upload page

This webpage handles uploading sets of DICOM files to the database with a Drag & Drop style interface.

We use the Flask extension Flask-Dropzone to create the Drag & Drop interface. The files uploaded are sent via a POST request to the /dropzone_handler URL. The files are then organized into an **ImmutableMultiDict**, which is a dictionary-like data structure defined in the Werkzeug package. This ImmutableMultiDict is found in request.files.

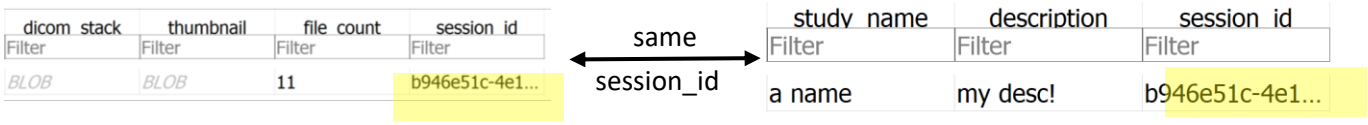
Although request.files is an ImmutableMultiDict, it can be treated like a traditional Python dictionary by calling **request.files.items()**. The key in this case is 'file', and the value is an instance of the **FileStorage** class (once again defined in Werkzeug)**.**

```
ImmutableMultiDict([('file', <FileStorage: 'D0016.dcm' ('application/octet-stream')>)])
```

Our code loops through the ImmutableMultiDict looking for 'file' keys and grabs the associated FileStorage objects to be added to the database.

Additionally, the upload page handles fields for *Study Name and Description* for each upload via the /form URL. Whenever a user navigates to the upload page a Session ID is generated that associates these two text fields with the files being uploaded. This acts as a foreign key in the database, so that we can keep these additional fields associated with their respective files.

DICOM files table **Form data table**



A global variable exists to handle the parallel upload of dropzone and how the files are upload in batches rather than as a whole submission. Counting the total number of files upload and appending the next batch of data uploaded.

The essential part of upload is the conversion of FileDataSet from pydicom into binary blob. To upload large files such as videos, it needs to be serialized or known as binary to compress data and transfer more easily in the database. Before the serialization we find the median image to generate a thumbnail .jpeg for the browser to display. The final part of code the calls **Dicom()** from **models.py** to insert the uploaded batch into the database.

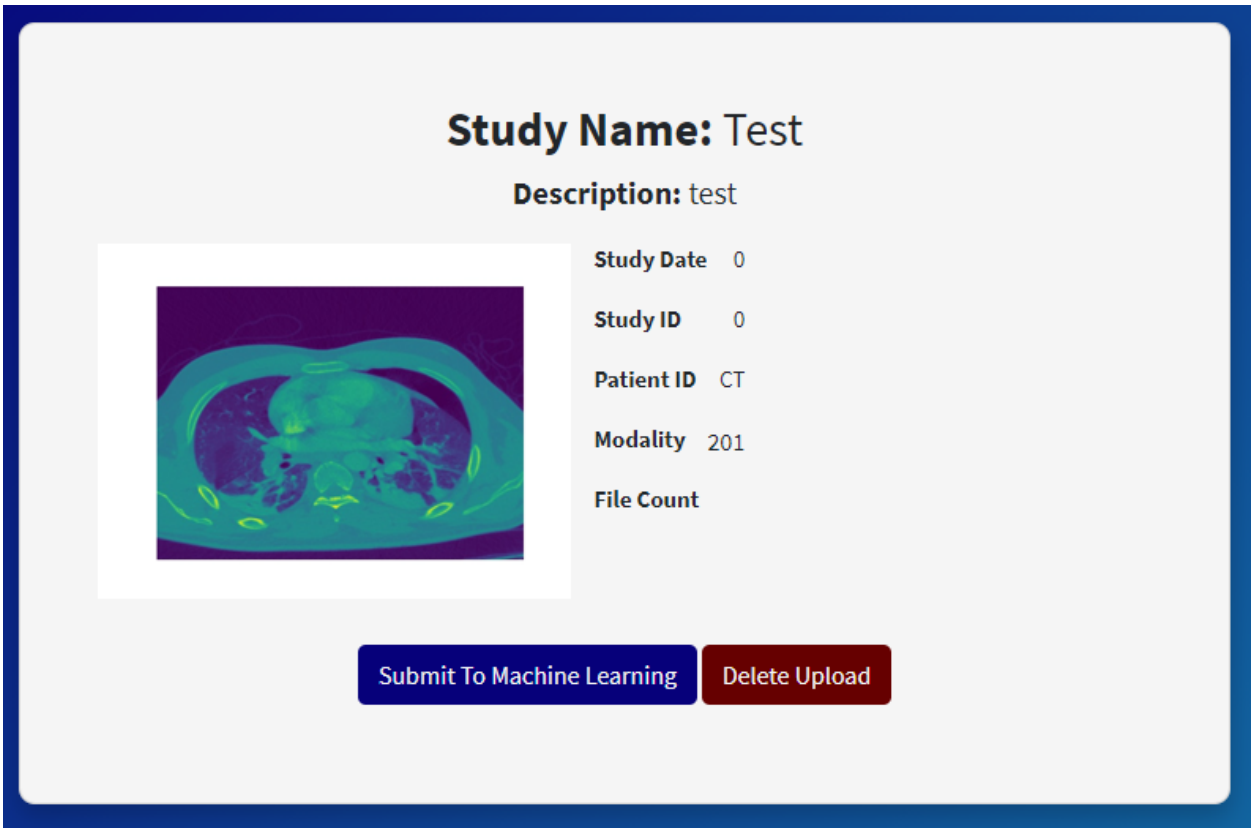
```
batch = Dicom(
    user_id=current_user.id,
    dicom_stack = binary_blob,
    thumbnail = tn_bytes,
    file_count= file_count_global,
    session_id=str(session['id'])
    )
```

Then the global variables are reset to default values.

Browser page

This webpage acts as a place for the user to browse through all their uploaded files and interact with them. From here they can run machine learning models with the selected files, or simply view individual DICOM files. The user can also view results of past models. Each set of files uploaded are paginated into a list-like fashion down the page with thumbnail images for each upload. Each thumbnail is a picture of the median slice in the set of DICOM files uploaded (the choice of the median file is arbitrary). Thumbnail images are stored in the database as raw data (bytes) and queried for display.

The user can click the 'submit to machine learning' to redirect to the 'job submission page' page to submit data to the Data Pipeline.



The browser page code is found in routes.py under file_pipline folder.

The database is query first; the DICOM files and the thumbnail is then deserialization back into its pervious object **FileDataSet**.

Job Submission page

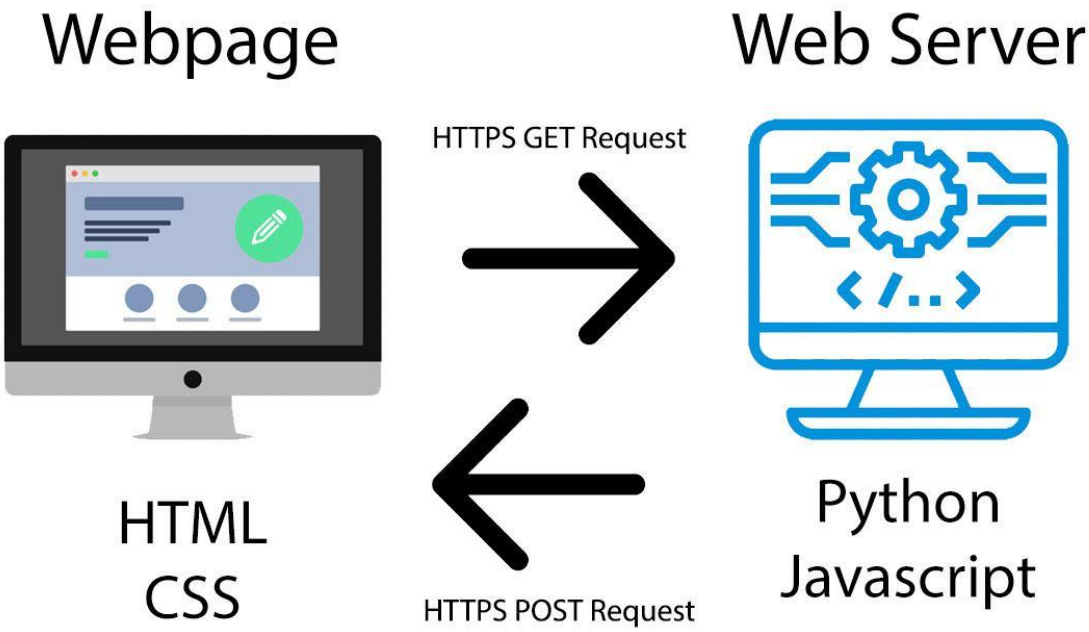
This webpage allows the user to submit their files into the Data Pipeline. Here they can specify how they want their machine learning model to run, ie changing various parameters of the model. The session_id generated on the upload page is passed to the job submission page, which is unique to the selected set of DICOM files to be processed. The session_id is used to query the selected DICOM files. The session_id, along with the other parameters set, is sent to the master (job queue), who then distributes the work to workers where the Machine learning models live.

3D Viewer page

This webpage is where the users can interact with the 3D model generated as a result of the machine learning model. When the model is generated, it is given a unique session_id_3d that identifies it in the database. This id helps us identify the

model if the user wants to access it again at another time. The brains of this page is the Visualization ToolKit (VTK). This powerful tool allows us to render 3D models in an interactive window. This involves a large amount of JavaScript to integrate this tool into the webapp.

How the Frontend interacts with Back-end server code



Back-end

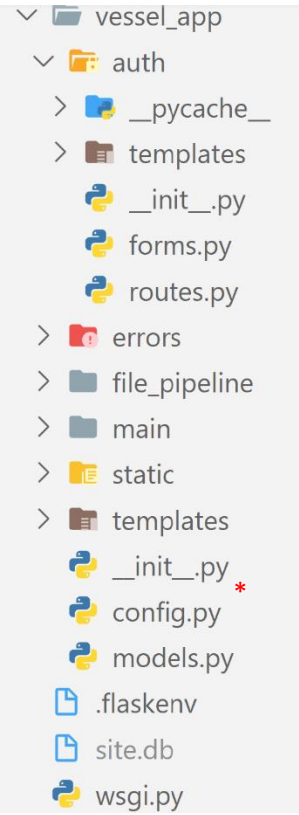
The functionality of the website (the Back-end) runs on the Flask app module (library) in Python. The Flask application, its code, its html pages, and requests are served on a web server called a uWSGI. The proxy server's specific function is to "manage web request from the internet concurrently to the web application". uWSGI is made specifically for python applications.

uWSGI references:

- Documentation - https://uwsgi-docs.readthedocs.io/en/latest/
- GitHub - https://github.com/unbit/uwsgi
- Issues - https://www.youtube.com/watch?v=p6R1h2Nn468
  https://uwsgi-docs.readthedocs.io/en/latest/ThingsToKnow.html
  https://uwsgi-docs.readthedocs.io/en/latest/Emperor.html

Project Files Structure

A flask app follows a specific file structure as seen in the photo below.



Blueprints

This specific style of organizing a flask project is called "blueprints".

The webapp is organized into several sections (blueprints), each contained in a folder. In the case of our project, we have 4 blueprints: auth, errors, file_pipeline, and main.

Each blueprint serves a specific purpose:

- **auth** – login, logout, and register functionality
- **errors** – 404, 500 error pages
- **file_pipeline** – Anything related to uploading, processing, and viewing DICOM files. The upload, browser, job process, and 3D Viewer page logic can be found here.
- **Main** – Any generic pages, such as the home page

As you can see in the expanded auth folder, each blueprint contains an **__init__.py, routes.py, and templates folder**. There is also sometimes a **forms.py** if appropriate.

The functionality of each of these files is as follows:

- **__init__.py** – Defines the blueprint instance and makes the folder callable as a python package.
- **routes.py** – Defines all of the URL endpoints that are associated with that portion of the webapp. This is also where the majority of the logic lives when it comes to processing files and the logic of how Front-end actually functions.
- **templates folder** – The templates folder contains all the relevant HTML files for that blueprint. The HTML files are not traditional HTML files, but rather have

special syntax in them that allows the Jinja2 engine to dynamically generate the webpages with information fed to them from routes.py.

- **forms.py** - A collection of classes that define forms. This is an alternative to the traditional html form, however it can make the process of creating more complex, secure forms easier. A forms.py file is built from classes as defined in the Flask extension.

Special Files

The master __init__.py

Each blueprint is useless by itself. In order to pull the blueprints together into a fully functioning website, we must have a file that connects all the blueprints into what is recognized as a complete website. To achieve this, we instantiate the application and all its dependencies (blueprints and Flask extensions) within the "context" of the overall application. This is done in a master __init__.py that is in the same directory as all the blueprints (denoted with a red * in the image above). **This key file is where all the parts of the application come together.**

config.py

This file's functionality is quite obvious. It is where all the configuration settings for various Flask extensions and other tools are defined. References to the database and celery broker URLs are also defined here.

models.py

This file is a collection of Python classes that generates the schema of the database. These models can be referenced throughout the different routes.py files in order to be able to query and write to the database. Some classes have special functionality that does not get relayed into the schema of the database, but rather caters to the object-oriented nature of Python. For example, the User class in models.py defines the schema for the User table in the database, but also has methods to create, validate, and encrypt (hash) passwords.

The static folder

This folder contains all of the JavaScript, CSS, image, and any other files that are unchanging, or "static", resources required by the webapp.

wsgi.py

This file, often called app.py in Flask projects, is where the webapp is run from. It calls the create_app() function as defined in __init__.py which runs the app on the specified port. In order to run the app a command prompt is opened in the same directory this file is located in and the "flask run" command is called.

OpenVessel's utility of JavaScript

To view the machine learning results on the CT scans in 3D, we use the npm module *VTK.js*. Because VTK.js is written in *ECMAScript 6* (also known as ES6, a more robust version of JavaScript not readable by a typical web browser), we needed to use *Webpack* and *Babel* to compile it into traditional browser JavaScript. This allows the web app to be run on any device.

JavaScript Environment

Our Node environment is found within the /static folder under /vessel_app . This includes our webpack configuration file and node package-json that is required to build and run ES6 code. The bundle.js file generated from webpack shares the /static/js folder with functional and webpage JavaScript.

To initialize the JS environment, open a terminal in the /static directory and run <code> npm install </code>. You'll need *npm* and *NodeJS* installed on your device.

To set up the JS environment for debugging, run the command <code> npm run watch </code> in the /static directory. This allows Webpack to automatically update bundle.js as changes are made.

Frequent issues:

- Make sure that the pathing is correct in webpack.config.js. Incorrect pathing could lead to a myriad of untraceable issues.

VTK.js viewer

VTK.js documentation: https://github.com/Kitware/vtk-js

Our 3d viewer can be found in */static/js/3d_vtk.js.* It receives a path to a VTI object in the server's file system and makes a interactable 3d viewer in the browser page's *#container* div.

For now, our system is to download the 3D object with the Flask app and then read it using a VTIReader function. We pass the path to the 3D object from the flask app to the HTML into 3d-vtk.js.

The viewer relies completely on the VTK.js library. It uses a pipeline methodology to process the VTI data into a discernable image. The necessary components are listed:

- genericRenderer - The actual viewport to see the 3d object
  - renderWindow - Interactable window that we see
  - renderer - Places the 3d object into a 2d space using a camera

- actor - A place for the 3d object to be filtered and processed
  - lookupTable - A function to color the 3d object based on different factors
  - piecewiseFunction - A function used to adjust the opacity based on different factors
- mapper - Takes the 3d object from the reader and processes it for the actor
- reader - Imports and parses a VTI object for the mapper to process

Since the reader uses an XML request to receive the VTI object, we use asynchronous JS (in the form of <code>.then()</code> statements) before dealing with the object's actual data. Any code outside simply prepares for the arrival of 3d data.

An essential part of rendering an appealing image is the widget. It uses the VTK widget preset "vtkVolumeController". (TO-DO)

Because we only have one JS function, our main js file addressed by Webpack (index.jsx) directly runs the viewer. Future development will focus on scalability in regards to JavaScript, including implementing React.js.

The Database – SQLite

We currently use an SQLite database.

Schema

The "models.py" script has classes or "models". These classes in python are used as a type of 'blueprint' or 'schema' that auto generates the SQL code necessary to model the schema of the database.

models.py imports a module called Flask-SQLAlchemy that allows Python and Flask app scripts to connect to any type of database and interact with it. The class model you see below uses function called **db.Column()** which tells the database to make a column with the defined datatype.

Commonly used datatypes with example uses:

- **db.Integer** - phone numbers, credit cards,
- **db.String** - usernames, emails, words in a blog post, passwords
- **db.DateTime** - today's date
- **db.LargeBinary** - images, videos, and DICOM files

```python
class Dicom(db.Model):

    ## data unqine id
    id = db.Column(db.Integer, primary_key=True)
    date_uploaded = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    dicom_stack = db.Column(db.LargeBinary, nullable=False)
    thumbnail = db.Column(db.LargeBinary, nullable=False)
    file_count = db.Column(db.Integer, nullable=True)
    session_id = db.Column(db.String(200), nullable=False)
    formData = db.relationship('DicomFormData', uselist=False, backref='author', lazy=True) #uselist one to one relationship


    def __repr__(self):
```

The class model above, "Upload_dicom" generates the SQL structure below. Every time a user uploads data it uses this schema to generate a table in SQLite.

```sql
CREATE TABLE dicom (
    id INTEGER NOT NULL,
    date_uploaded DATETIME NOT NULL,
    user_id INTEGER NOT NULL,
    dicom_stack BLOB NOT NULL,
    thumbnail BLOB NOT NULL,
    file_count INTEGER,
    session_id VARCHAR(200) NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES user (id)
)
```

CRUD Methods

When a user inputs login information, registration information, or uploads DICOM files, they are added to the database. They exist in these tables and the web application uses these schemas to **CREATE** data tables, **RETRIEVE** data from data tables, **UPDATE** data in tables, and **DELETE** data. This is known as the CRUD model. Nearly every web application uses this functionality. CRUD is just describing the action between the webapp and the database.

Flask-SQLAlchemy allows us to **create** organized tables with the specified schema in our database. From there we can **update** the table with the files provided by the user. When the user wants to process their uploads, we **retrieve** the queried data to display in the browser webpage. We can also **delete** the specified files. We can finally push the retrieved data to the Data Pipeline via a job queue.

Foreign Keys

In addition to being able to apply the CRUD methods to our database, we can also establish relationships between tables using "foreign keys". This is a shared value between two related rows in two different tables. For example, we can specify what user created each set of DICOM files in the database by referencing their user ID. Therefore, there is a shared value in each table that connects them logically.

The relationship between one user in the User table to many rows in the DICOM table is known as a **one to many** relationship.

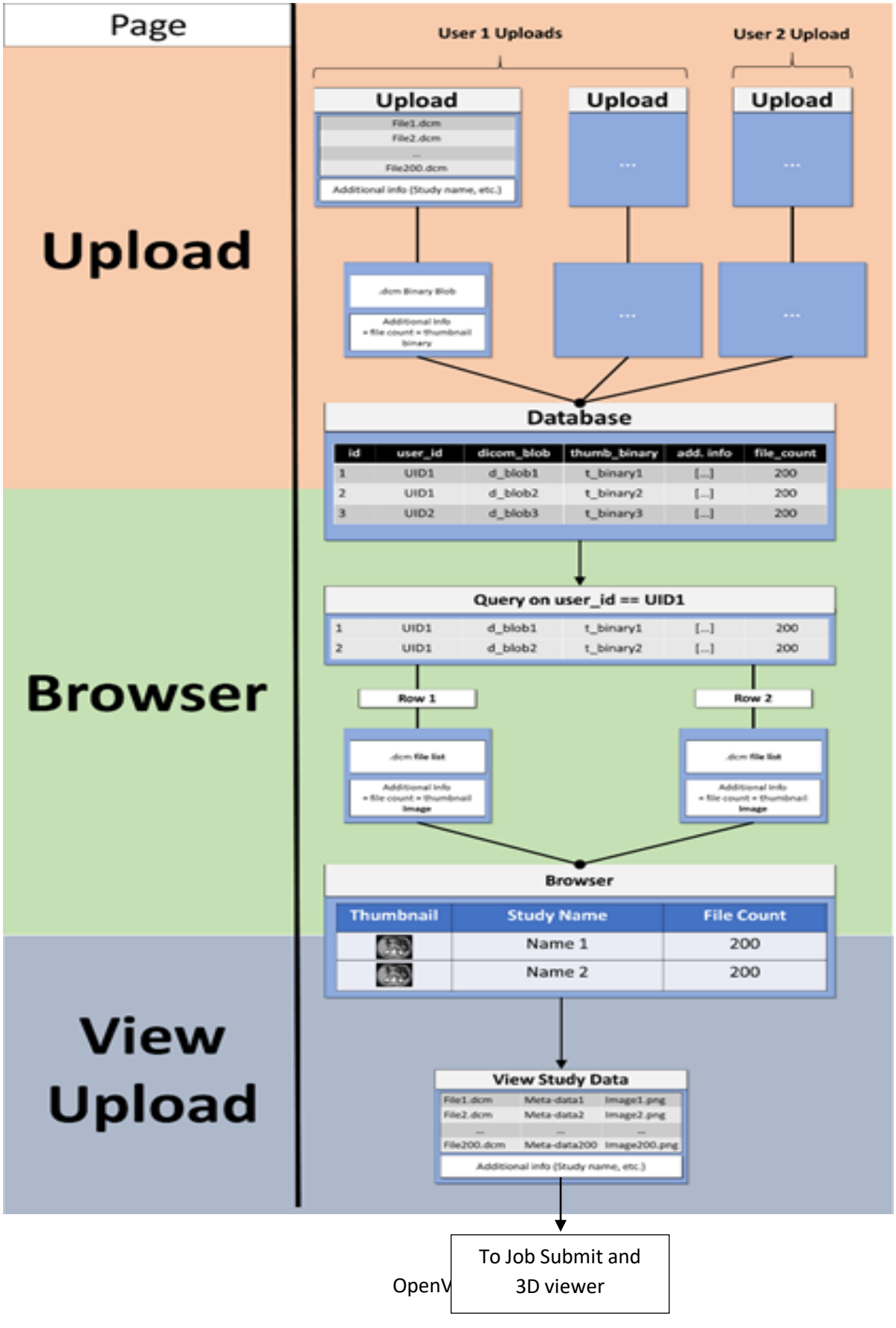
User Table

id	username	email	image_file	password
Filter	Filter	Filter	Filter	Filter
1	test2@gmail.c...	test2@gmail.c...	*NULL*	$2b$12$pqXLl...

DICOM Table

id	date_uploaded	user_id	dicom_stack	thumbnail	file_count	session_id
Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	2020-07-27 1...	1	*BLOB*	*BLOB*	6	930f8dda-78e...
2	2020-07-27 1...	1	*BLOB*	*BLOB*	12	990363ba-2d...
3	2020-07-27 1...	1	*BLOB*	*BLOB*	6	5f06205d-365...

The database in the context of the webapp



Distributed Computing - Celery

In order to minimize the time, it takes to run the machine learning models, we use a distributed computing model. This means that the workload coming in from various users is distributed across multiple workers instead of the job queue being backed up into one worker. This model is implemented through the Celery-Kombu Python package.

Due to the complexity of the Celery package, the process of initializing celery is different than that of other tools/Flask extensions. Inside __init__.py there is a separate function that instantiates Celery apart from the rest of the application.

```
def create_celery_app(app=None):
    app = app or create_app(Config)
    celery = Celery(__name__,
    backend=app.config['CELERY_RESULT_BACKEND'],
    broker=app.config['CELERY_BROKER_URL'])
    celery.conf.update(app.config)
    TaskBase = celery.Task

    class ContextTask(TaskBase):
        abstract = True

        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)

    celery.Task = ContextTask
    return celery
```

2

1

There are a couple things to note here:

1. We modify the __call__ function of the celery task to be called within the context of the Flask app.
2. We pass the "CELERY_RESULT_BACKEND" and "CELERY_BROKER_URL" variables into the constructor. Both of which are built off a **Redis server**. These variables are defined in config.py

Some useful definitions related to Celery and distributed computing as a whole:

- **Task Queue** - A system for parallel execution of discrete tasks in non-blocking fashion.
    - All versions of an application share the queues
    - Push queues for auto execution
    - Pull queues to pragmatically consume tasks
- **Task** - A unit of work such as 'write object to datastore' or 'send an email'.
- **Producer** - The code that places the tasks to be executed later in the broker – application code.
- **Worker**- Takes a task from the broker and performs it.
- **Broker** - The middleman holding the tasks (messages) themselves.

The Broker

The broker runs on Redis by default and requires live Redis to pass and process message between the web application and the worker pool. The broker can be reconfigured in **config.py**.

```
### Celery Brokers ##########
CELERY_BROKER_URL='redis://localhost:6379/0'
CELERY_RESULT_BACKEND='redis://localhost:6379/0'
```

Types of brokers:

- Synchronous – The caller waits for a response before sending the message. Basically, the web application sends a message to the broker and the broker sends back a response.
- Asynchronous - The message is sent without waiting for a response. This is best for distributed systems I.E. our system.
    - Asynchronous Broker considerations:
        - Broker Scale – The number of messages sent per second in the system
        - Data Persistency – The ability recovery messages
        - Consumer Capability – One to one or one to many consumers

Question to ask yourself about the Broker.

- How you structure microservices?
- Infrastructure?
- Latency?
- Scale?
- Dependencies?
- The purpose of communication?
- Broker central discovery?
- Monitoring?
- Load balancing?
- Policy enforcement?

In **routes.py** an blueprint call is made as shown. You can make many more blueprint calls in flask to create multiple celery for separate application within the system.

```
@bp.route('/3d_viewer', methods=['POST'])
def viewer_3d():
```

Here we call worker a **task** to be processed.

```
# Call worker and save result to database
result = data_pipeline.delay(session_id, session_id_3d, n_clusters=k)
result_output = result.wait(timeout=None, interval=0.5)

#test_data = load_data.delay(session_id)
#print(test_data)

## WORKER CALL CHain workflow
# celery.chain(query_db_insert() ,
# load_data(),
# resample(),
# lung_segmentation(),
# pyvista_call(),

# ).apply()
```

Data Pipelines

The Data Pipelines are the core of the project. Each "pipeline" is a machine learning model. The user will have the ability to switch between pipelines in order to run different models.

NOTE: The Masking Segmentation Pipeline is currently the only working pipeline. It needs optimizations and a module called VTK and pyvista that uses C++ algorithms and Python to make the overall pipeline faster.

The worker task and all functions calls are coded in a sequence in celery_tasks.py in data_pipeline().

Available Pipelines:

- Masking Segmentation Pipeline

The web application generates a "message" containing the session_id that is passed to the celery task

1) The database is query based off generated session_id
2) The data is deserialized – converted from binary to its original file format
3) **load_scan()** converted back into dicom files
4) **get_pixels_hu()** converts pixel data into Hounsfield units
5) **resample()** the data for volume process time
6) **make_lungmask_v2** applies K-means algorithm to each slice.
7) **displayer()** – calls python wrapping pyvista that utilizes VTK to convert the NumPy array into 3D object- data type known as uniform gridded.
8) The 3D object is serialization into binary and the BLOB is inserted back into the database

The conclusion now the web application can query the 3d object from anywhere across the web.

Data Preprocessing before Machine Learning

These are 3 basic functions needed to convert a DICOM file into an object known as FileDataset in the pydicom package. The FileDataset object has attributes to call specific metadata, such as the pixel data.

- Load_scan() - loads the dicom file in an object called FileDataset generated by dcmread function from the pydicom module.
- Get_pixels_hu() - pixel data is convert to HU
- Resample() -

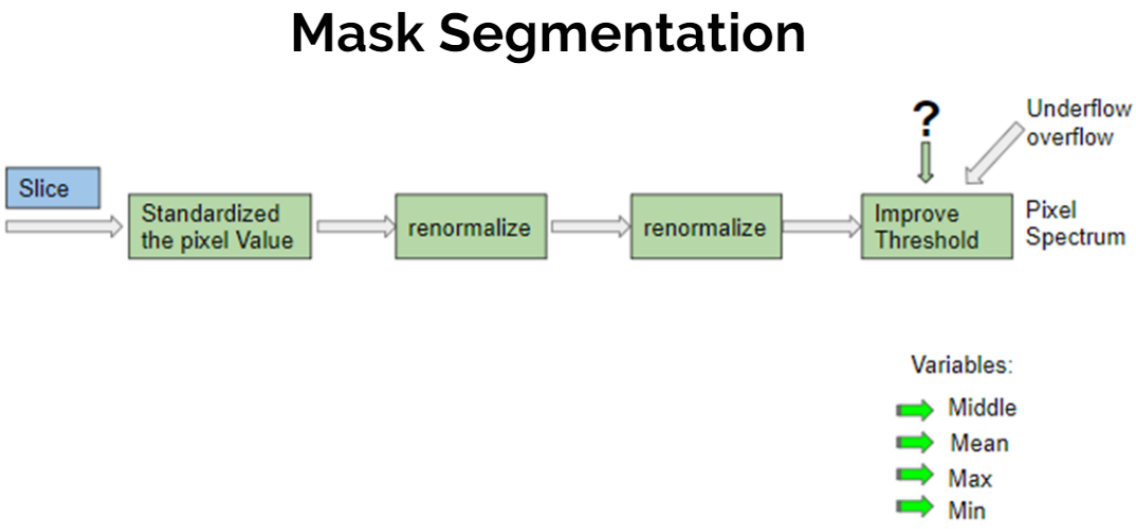
Load Scan

Pixel conversion to HU

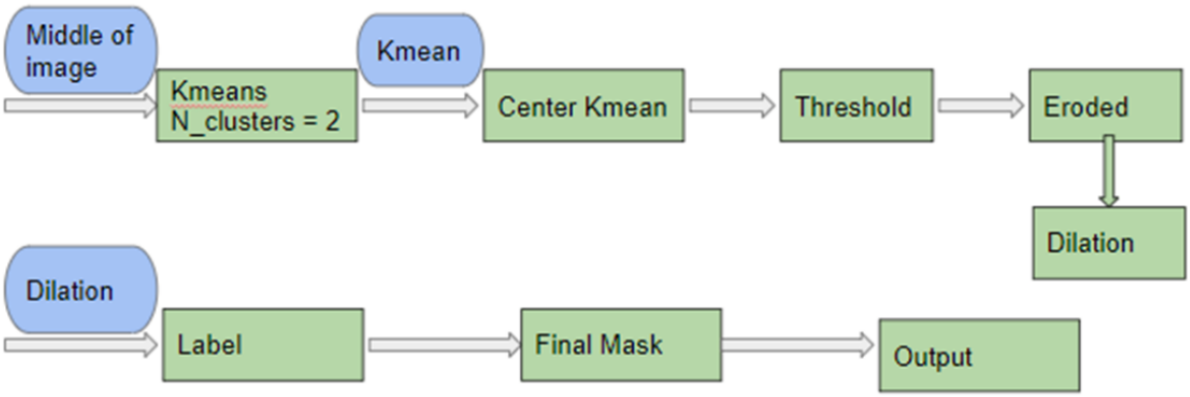
Resampling

The metadata in DICOM images called 'Slice Thickness' is used to adjust and resize voxel data so the data can be used by machine learning algorithms. Resampling resolves this issue by making all the data 1 mm × 1 mm × 1 mm which reduces the number of data points to be processed.

Masking Segmentation Pipeline

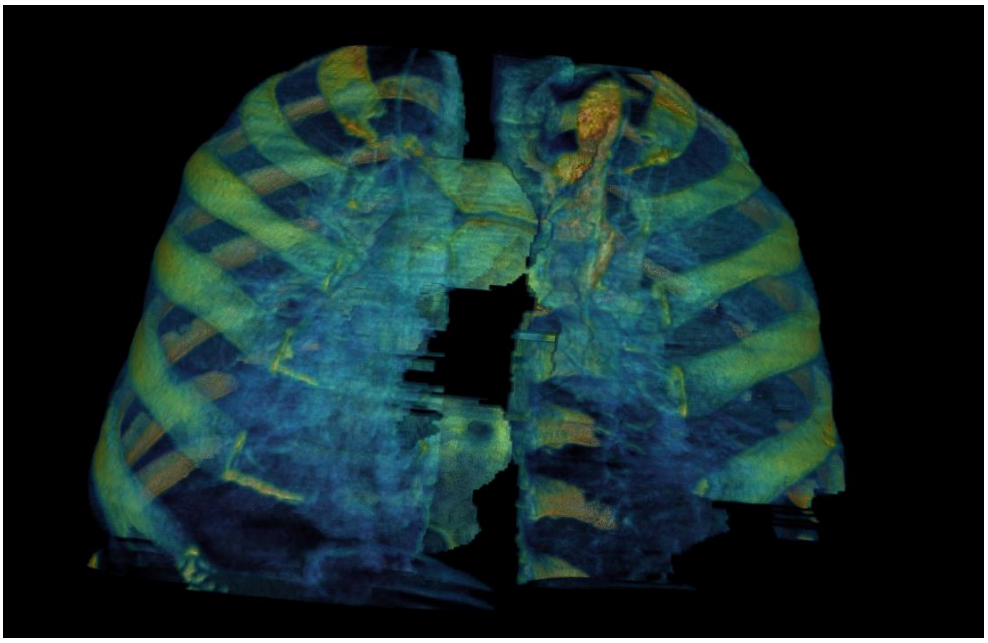


Mask Segmentation (Continued)



VTK.js and the 3D displayer

A 3D image of the scan is generated by applying a marching cube lewiner algorithm to generate a mesh over the 3D object. Further segmentation can be applied to the lung data for patient case use. The application of region growing and morphological operation can be applied to improve results. The easy way is to use the connected component analysis to determine what voxels are air and what voxels are vessels or lung tissue.



How VTK works for OpenVessel

Our machine learning models generate 3D array mask or binarny mask. 3D numpy arrays.

PyVista wonderful python wrapper around VTK C++ simplify its data structures.

Work Cited

[1] PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK) C. Bane Sullivan1 and Alexander A. Kaszynski2 1 Department of Geophysics, Colorado School of Mines, Golden, CO, USA 2 Universal Technology Corporation, Dayton, OH, USA

[2] https://www.bu.edu/tech/support/research/training-consulting/online-tutorials/vtk/

[3] https://vtk.org/Wiki/VTK/Presentations

[4] https://www.evl.uic.edu/aej/524/lecture03.html

[5] https://serc.carleton.edu/NAGTWorkshops/data_models/toolsheets/vtk.html

[6] https://arxiv.org/abs/1606.06650

[7] https://github.com/ellisdg/3DUnetCNN

[8] https://www.fullstackpython.com/postgresql.html