MIDI Validation - User Manual

The MIDI validation script is used to validate curated datasets to ensure they have been de-identified according to both DICOM and TCIA Best Practice Standards.

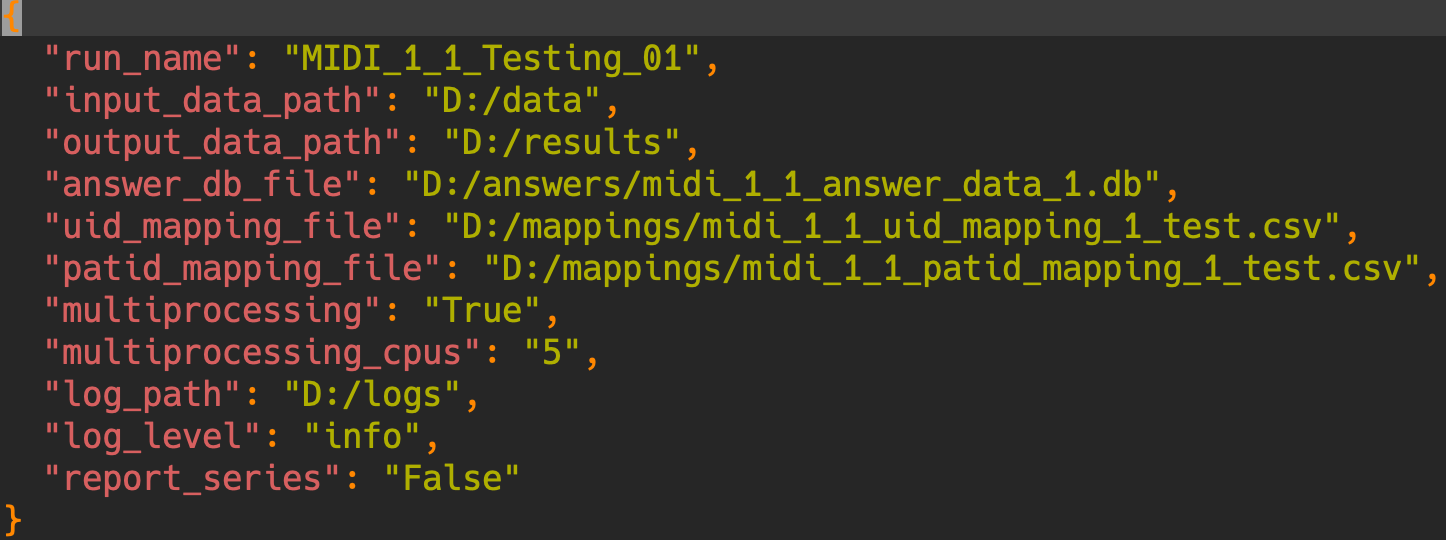
Disclaimer: This validation is not 100% about being wrong or right. It is simply a recommendation to guide the testing of de-identification processes. While some of the “answers” in the answer key will adhere to strict DICOM standard requirement guidelines, many are simply marked according to how TCIA handled it when the collection from which the base image was retrieved, was originally curated.

# Configuration File

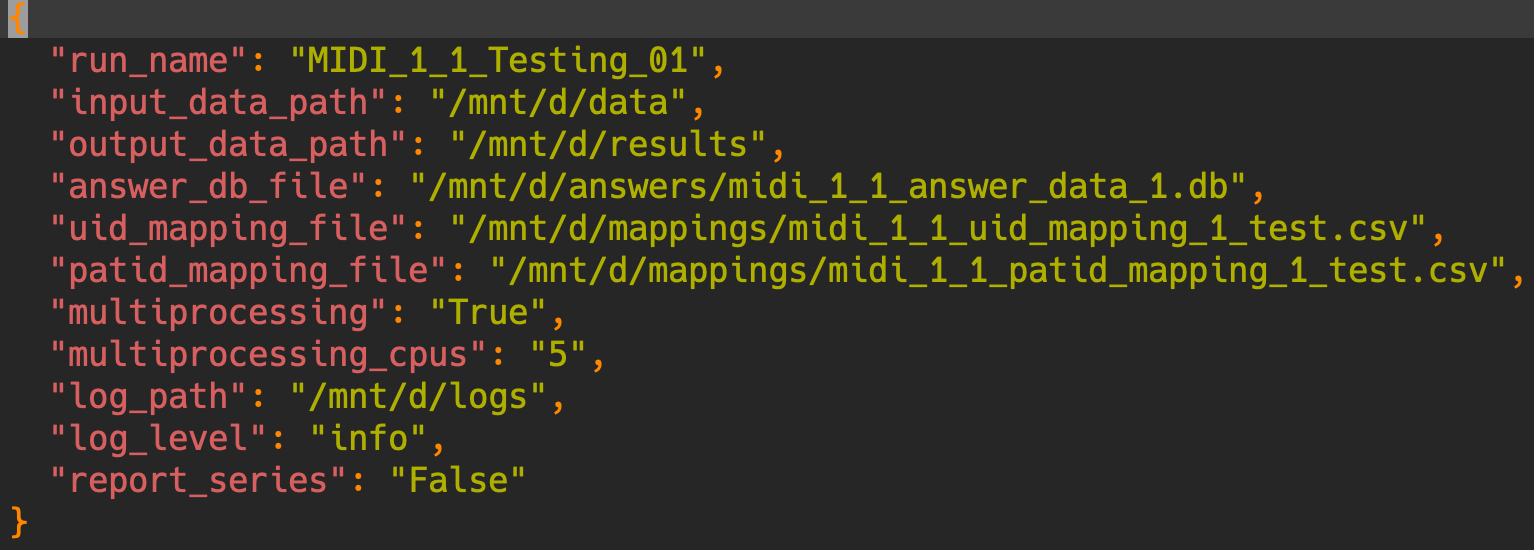
The main component that drives all functions in the validation script is the configuration file. The configuration file is a JSON file that contains multiple parameters that can be customized to your personal environment.

Both Windows and Linux paths can be used in the configuration file depending on the OS you plan to run on.

The following is an example of a configuration file for Windows (config\_example\_win.json)



The following is an example of a configuration file for Linux (config\_example\_linux.json)

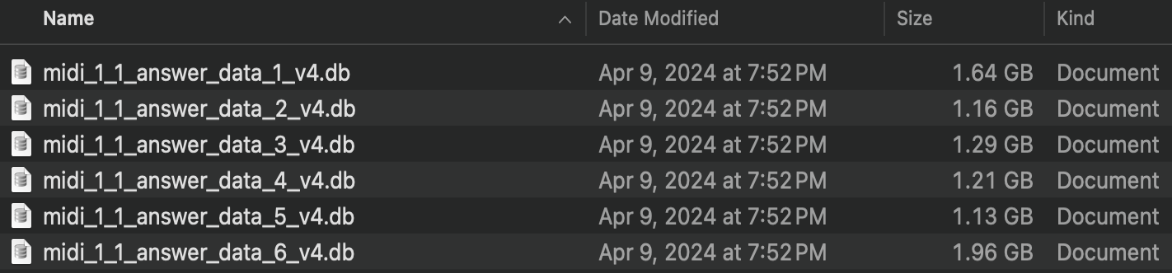


The parameters are as follows:

* **run\_name** – The name of this validation instance. This folder will be created in the output path below and used to organize all files for this validation run. *NOTE*: reuse in other config can overwrite if same folders are used.
* **input\_data\_path** – This is the path to the images to be validated.
* **output\_data\_path** – This is the path where you want all output files to be placed. The “run\_name” folder will be placed at this path and used for all run files.
* **answer\_db\_file** – This is the path to the SQLite database containing the answer key data being validated against.
* **uid\_mapping\_file** – This is a path to the UID mapping file. This file contains a cross reference of all study, series, and instance UIDs which map the old set to the new set.
* **patid\_mapping\_file** – This is a path to the patient id mapping file. This file contains a cross reference of patient ids which map the old set to the new set.
* **multiprocessing** – “True”-”False” toggle to use multiprocessing.
* **multiprocessing\_cpus** – The number of CPUs to dedicate to multiprocessing.
* **log\_path** – The folder path for storage of execution logs.
* **log\_level** – The lowest log level shown in logs. Can be debug, info, warning, error, critical.
* **report\_series** - “True”-”False” toggle to report by series

# Answer Key

The answer key data is stored in SQLite databases. The answer key contains individual records for all images along with the corresponding answer records in JSON format for consumption by the validation script. The MIDI 1.1 validation package is composed of six test data sets, and six answer keys. Folder view below.



These database files can be viewed using various tools or code, but for casual browsing we recommend [DB Browser for SQLite](https://sqlitebrowser.org/). Using this software, the “Browse Data” tab (screen below) will enable a spreadsheet-like view of the data which can be sorted and filtered.

Graphical user interface, text, application

Description automatically generated

# UID Mapping File

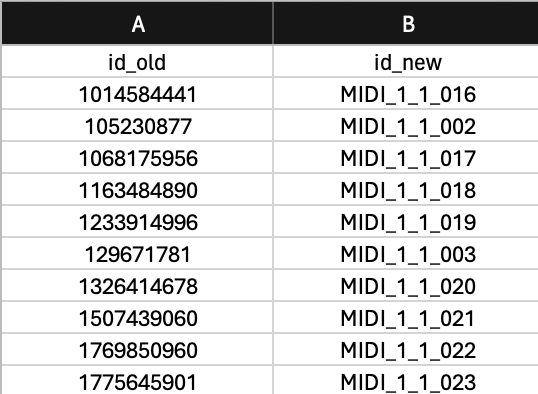
The UID mapping file contains the old (test dataset) and new (curated dataset) study, series, and instance UIDs for the collection. This file is required, else, the validation script cannot match images to answer key entries. The mapping file should look similar to the image below, using the same column headers, “id\_old” (test) and “id\_new” (result).

Table

Description automatically generated with medium confidence

# Patient ID Mapping File

The patient id mapping file contains the old (test dataset) and new (curated dataset) patient IDs for the collection. This file is required to check the patient ID consistency. The mapping file should look similar to the image below, using the same column headers, “id\_old” (test) and “id\_new” (result).



# Validation

Text

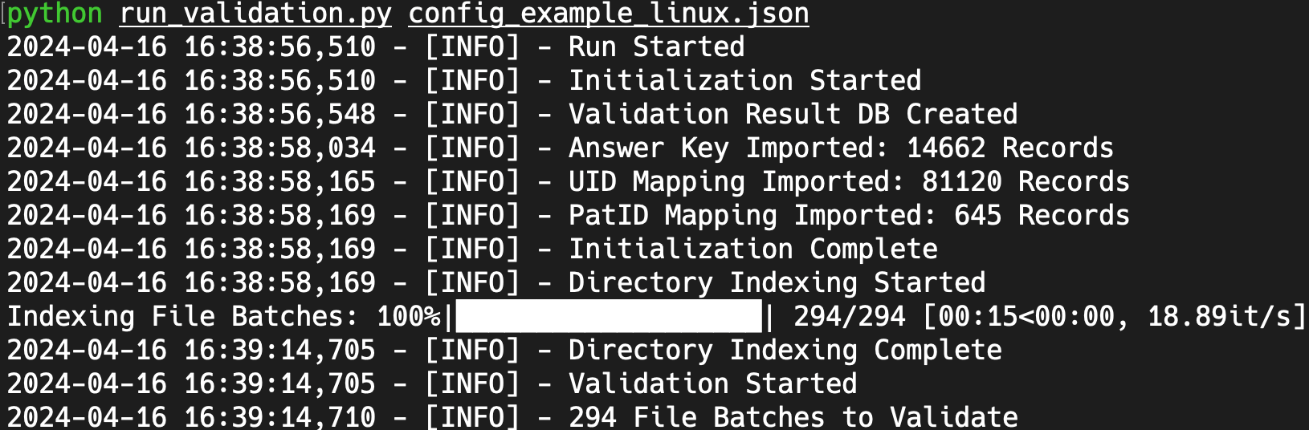
Description automatically generatedThe validation has evolved since the previous version. It is more modularized and more informational in its logging capabilities. The image at right displays the python scripts and files in this package. At the root of the directory is the requirements.txt file and three main “run\_” scripts. All modules used by the run scripts are housed in the “modules” folder and should not be ran independently.

## Requirements

Python 3.8 is required to run this validation script. In addition, the requirements.txt file contains a list of all Python packages and versions required to be installed.

## Execution

The validation can be started using the “run\_validation.py” script. This script requires a single parameter, the configuration file path (or filename if located in the script root folder).



This will display a series of steps while performing the validation as displayed above. Please note that this information may be displayed mixed between different modalities when multiprocessing is enabled. This information will also copy to the log file at the location specified in the configuration file. When an error occurs as is displayed below, it will not disrupt the execution, but it will log the action, file path, instance UID, and the tag being checked as displayed below. Each log file is unique to each execution run. It is recommended to search for “[ERROR]” in the log at the end of an execution to ensure no major errors occurred.

Text

Description automatically generated

The output of the validation run will be two files (folder view below). “validation\_results.db” containing all the validation checks and their results, and “pixel\_validation.xlsx” listing burn-in pixel data which must be validated. We will cover this in more detail in the next section.

Graphical user interface, text

Description automatically generated

## Import

*Important*: The import process must use the same configuration file used in the validation or errors can occur.

This import process involves the pixel\_validation.xlsx file which was the output from the previous validation step. The file contains information about burned in pixel data which needs to be validated (image below). The main values of interest are the file path, used to open the file, check\_passed, and check\_score.



The check\_passed and check\_score variable should be edited in this step.

* **check\_passed** – validation passed. Enter **1** for *True* (Text obscured), **0** for *False* (Text Present).
* **check\_score** – percentage correct. Value between 0 and 1. Ex. If half the text still visible, you would enter 0.5.



In the image above, the text was present so 0 is entered in both fields.

*Important*: No other values should be edited in this file, especially the first column, an index that links back to the database record. When all records are validated, save and run the import script.

Similar to the execution of the validation script, the import script requires the config file name. (Screen below)

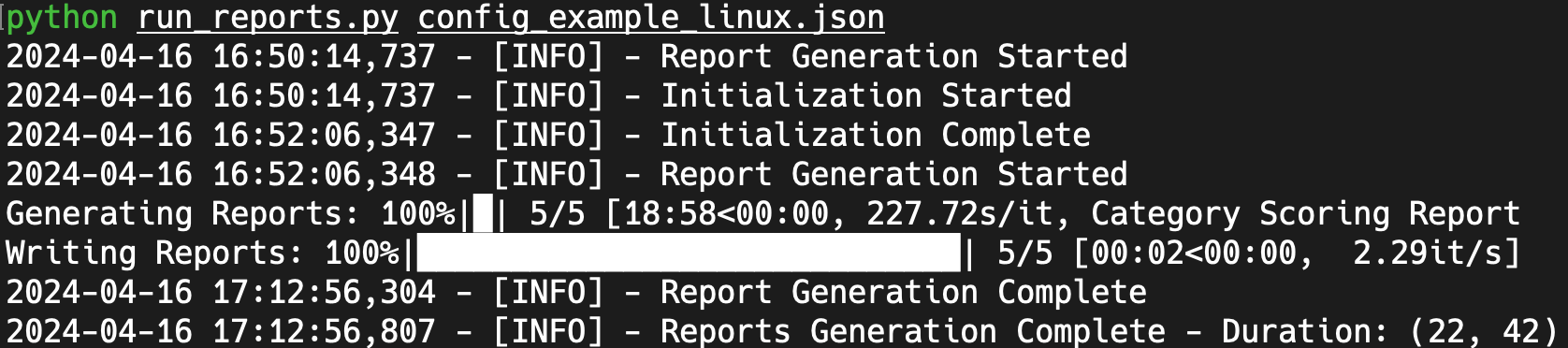
Text

Description automatically generated

# Reporting

*Important*: The reports process must use the same configuration file used in the validation or errors can occur.

Reporting is the final step of the validation process. This process creates two spreadsheets, the discrepancy report and a scoring report with multiple tabs (Scoring, Action, and Category). As with previous, the reports are run with the configuration file name as the only input. (See syntax below)



## Discrepancy Report

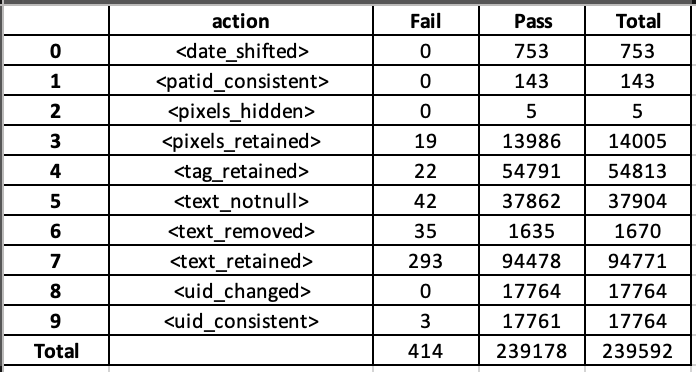
The discrepancy report provides a report of all validations that did not pass.

Table

Description automatically generated

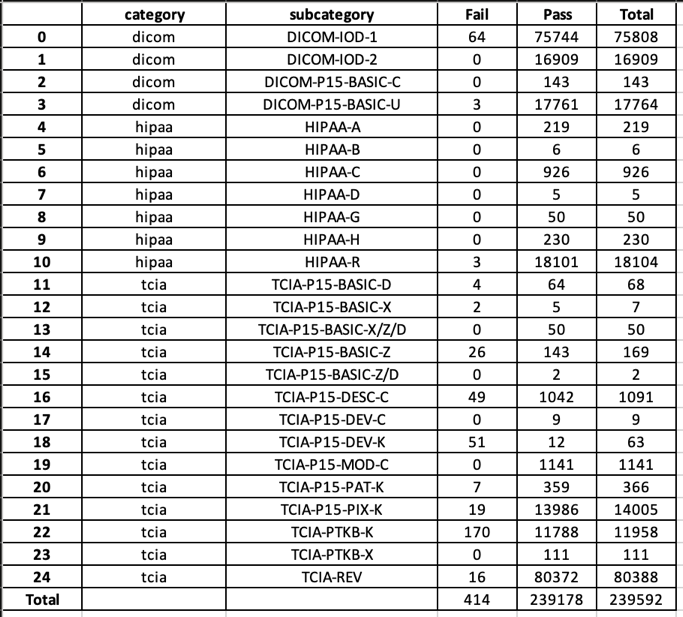
## Action Report

The action report provides counts of Passed and Failed values per validation action.



## Category Report

The category report provides counts of Expected (passed) and Unexpected (failed) values per answer category. The categories were created to accommodate the scoring system.



## Scoring Report

The scoring report takes the categories from the previous report and assign them to scoring categories. The score is calculated in terms of the ratio of fail/pass actions, which are described in the previous section.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Fail** | **Pass** | **Total** | **Score** |
| All | 414 | 239178 | 239592 | 99.83% |

## DCIODVFY Report

The DCIODVFY report is generated from running David Clunie’s DCIODVFY software on the dataset to assess compliance to the DICOM standard. [Dicom3tools](https://www.dclunie.com/dicom3tools.html) is a required package for executing the command of ‘python run\_dciodvfy.py config\_example\_linux.json’. Dicom3tools also can be installed using “sudo apt install dicom3tools” on Linux.

|  |  |  |  |
| --- | --- | --- | --- |
| **type** | **tag** | **message** | **modality** |
| <Warning> | </SeriesNumber(0020,0011)> | <Missing attribute or value that would be needed to build DICOMDIR> | <RTPLAN> |
| <Warning> | </InstanceNumber(0020,0013) > | <Missing attribute or value that would be needed to build DICOMDIR> | <RTPLAN> |
| <Warning> | </PixelData(7fe0,0010)> | <Attribute is not present in standard DICOM IOD> | <RTPLAN> |
| <Error> | </Laterality(0020,0060)> | <Missing attribute for Type 2C Conditional - Module=<GeneralSeries>> | <MR> |