Documentation for the MIMIC-III Dataset Generation Library (MDGL)

INTRODUCTION

This report contains the documentation for the Mimic-III Dataset Generation Library (MDGL). MDGL is an effort to both reduce the effort needed to quickly obtain patient datasets from the Mimic-III database and allow for increased consistency in data used across different researchers. To further facilitate the widespread use of MDGL, all of the code has been written in the Python Version 2.7 language.

The report will consist of three main sections: Specifications, Library, and Setting Up and Running. Section 1 details the specifications that a user apply to how the dataset is generated. Section 2 provides a thorough documentation of all code files, the functions within, and how to expand them. Lastly, Section 3 covers additional tools built on top of the MDGL, specifically the feature-frequency cluster analysis tool.

SPECIFICATIONS

The following section will describe the features that a user can specify for generating a patient dataset, along with how the syntax needed to specify all features.

SPECIFICATIONS FILE

All user specifications should be created within a single file, e.g. `Specifications.txt`. This file should reside within the directory that contains the code for the MDGL. When the code is run, it will search for the specifications file specified by the user in the current directory and parse it; if this file is not found, an error will be thrown.

If a user wishes to store multiple specification files, this is not a problem. When a patient dataset has been created, a copy of the specifications file used to create that dataset will be placed within the patient dataset directory. This will ensure that all specifications files can be kept track of and shared if another user wishes to reproduce the generated dataset.

SPECIFICATIONS FORMAT

All specifications within the specifications follow must follow a strict, yet simple, format. Currently, there are three main sections within the specifications file: ICU section, patient section, and the parameter section. An example for how the specifications file should look can be found in the Appendix. Furthermore, if the user wishes to add text before or after the specifications in the specifications file, the tags `#Begin` and `#End` can be used before and after the specifications.

ICU SECTION

The ICU section allows a user to specify which of the six ICU types in Mimic that they wish to include patients from in the generated dataset. These ICU types are CCU, SICU, MICU, NICU, CSRU, and TSICU. The ICU section should start with the line:

```
#ICUs
```

In order to specify which ICU types to consider, the following line should be added beneath the `#ICUs` tag:

```
[ICU][True|False]
```

where `[ICU]` is one of the six ICU types, `True` indicates that the ICU type should be included, and `False` indicates that the ICU type should be excluded. Below is an example of what the ICU section in `Specifications.txt` may look like if a user wishes to include ICU types CCU, SICU, MICU, and CSRU and exclude ICU types NICU and TSICU:

```
#ICUs
CCU    True
SICU   True
MICU   True
NICU   False
CSRU   True
TSICU False
```

Patient Section

The patients section allows a user to specify what range of ages (in years) and the sex of patients that should be considered. It also allows the user to specify how many hours of an ICU stay should be used. The patient section should start with the line:

```
#Patients
```

Age

To specify the age range, the following line should be added beneath the `#Patients` tag:

```
Age; [min]; [max]
```

where `[min]` is the minimum age in years to be considered and [max] is the maximum age in years to be considered. Below is an example of how a user can specify that they only wish to use patients between the ages 16 and 21:

```
Age;16;21
```

If a user wishes to consider all ages above a certain age, the `[max]` argument can simply be replaced by a large enough number, even one such as 999. Likewise, if a user wishes to consider all ages below a certain age, the [min] argument can be set to 0. Thus, if a user wishes to use all ages, the user can set `[min]` to 0 and `[max]` to 999.

Sex

To specify the sex to be used, the following line should be added beneath the `#Patients` tag:

```
Sex; [sex]
```

where `[sex]` is used to specify the gender. The argument `[sex]` can be set to `M` to consider only males, `F` to consider only females, or `Both` to consider both males and females. An example for a how a user can specify to use both genders is shown below:

```
Sex; Both
```

Hours

To specify how many hours of an ICU stay to use, the following line should be added beneath the `#Patients` tag:

```
Hours; [hours]; [required]
```

where `[hours]` should be a numerical value greater than 0, and `[required]` should be either a 0 (patients can have any number of hours in the ICU) or a 1 (patients must have at least `[hours]` number of hours in the ICU). All recorded measurements for a patient's ICU stay (across all included ICU types) from when they entered the ICU up until the specified number of hours will be included in the dataset.

Parameter Section

The parameters section allows a user to specify which numerical measurements should be included in the dataset. The measurements will be extracted from any of the CHARTEVENTS, LABEVENTS, and OUTPUTEVENTS tables in Mimic. For each measurement that should be included in the dataset, the following line should be added beneath the `#Parameters` tag:

```
[name]; [description]; [[IDs]]
```

where `[name]` is the name or abbreviated name for the measurement that will be used to identify that measurement in the patient output files, `[description]` is the full name or text description of the measurement (mostly useful as a side note), and `[IDs]` is a comma-separated list of all Mimic itemids that specify the measurements in the Mimic database. An example for how a user can specify measurements is provided in the Appendix.

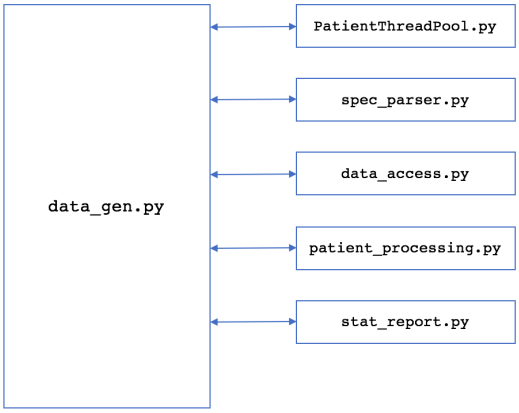

Figure 1: MDGL Software

Library

The MDGL currently consists of six separate scripts: `data_gen.py`, `data_access.py`, `patient_procesing.py`, `spec_parser.py`, `stat_report.py` and `PatientThreadPool.py`. Figure 1 shows how these files are related. The main file that is run is data_gen.py, and the other files contain related functions that are used by data_gen.py. The sections below will list code dependencies, describe the output, and detail the functionality assigned to each of the scripts and any other related information.

Dependencies & How to Run

In order to use the MDGL, the `psycopg2` and `numpy` libraries need to be installed or placed within the current working directory.

The user also needs access to an instance of the Mimic-III database, along with a username and password for access. Once these libraries have been installed and an instance of Mimic is available, the MDGL can be run using Python 2.7 with the command:

```
python data_gen.py [host] [port] [specfile]
```

where `[host]` and `[port]` refer to where the database is located (if local, should be **`localhost 5432`**), and `[specfile]` is the name of the user's specifications file.

OUTPUT

The output will be a directory named "`patientfiles [YMD-HMS]`", where `YMD-HMS` is the year, month, day, hour, minute, and second at which the directory will be created. All of the patient files, the specification file, and the report will be located in this directory.

Regarding the format of the patient files, each line has the following format:

`[HH:MM], [name], [ID], [value]`

where `[HH:MM]` is the time based on the start of the stay (potential values range between 00:00 to XX:00, where XX is the maximum number of hours specified in the specifications file), `[name]` is the name of the measurement (see `[name]` in the parameters specification above), `[ID]` is the specific item ID of the measurement (which maps the measurement directly back to Mimic), and `[value]` is the numerical value of the measurement.

The first line of a patient file contains the column headers. The next six lines contain static information about the patient, which includes the patient's record ID, age, gender (0 for female, 1 for male), height, ICU type (0 for CCU, 1 for SICU, 2 for MICU, 3 for NICU, 4, for CSRU, 5 for TSICU), and weight at ICU admission. All lines after the first seven lines contain measurement information collected from the database.

DATA_GEN.PY

As mentioned above, `data_gen.py` is the main file that should executed. It initializes certain data and classes needed throughout the patient generation process, and it imports the functionality of the other files. The overall patient generation process is very step-driven, with each library being responsible for a certain step.

This script will start by establishing necessary connections to the Mimic-III database. This includes setting up a single connection as well as initializing an instance of the `PatientThreadPool` class that will set up connections as well.

PATIENTTHREADPOOL.PY

This file `contains` the class `PatientThreadPool` that implements a thread pool that can be used to parallelize functions across all available cores, namely those that access the Mimic database. This is because the `PatientThreadPool` class initializes a database connection for each virtual core, allowing each thread to access the Mimic database concurrently. The main advantage that this class offers is to offer significant speedup through a flexible class that can easily be expanded.

The decision to use the python `Threading` library over the python `Multiprocessing` library came about as follows. When using the `Multiprocessing` library, different library imports, such as `numpy`, affect the core affinity and cause the processes to only run on a single core. Furthermore, when using the `Multiprocessing` library's `Pool` class, that acts as a thread pool, it attempts to split up the arguments equally among all processes. However, as larger arrays of data are passed in, issues were encountered with the argument splitting taking much too long compared to what could be done manually. Finally, it makes sense that threads are used instead of processes so that data can be shared as needed.

One potential problem with this approach is that that Python's global interpreter lock can limit all threads to work on only a single core. Although this is only a problem in certain scenarios and does not affect MDGL, it should be kept in mind in with future MDGL development.

To use the `PatientThreadPool` class, it must be initialized by passing it the username and password necessary for connecting to the Mimic database.

Afterwards, the function `executeFunc` should be called, passing in the function to be executed, a list of arguments that all instances share, and a list of arguments that need to be split equally among all threads. `executeFunc` will automatically handle argument splitting and thread initialization and termination. The cumulative result from the threads can be accessed by calling the class function `getResults`.

In order for a function to be called by the `PatientThreadPool` class, it needs to adhere to two simple specifications.

First, it should only accept as input a single object. This object will be the entire arguments array, passed in by `executeFunc`. The first few arguments will be those that the user passed in `args`, the next few arguments will be those that the user passed in through the split `args`, the second to last argument will be a reference to the thread pool class, and the last argument will be a database connection (if needed).

Second, instead of returning a value, it should acquire the thread pool's lock, append its result to the thread pool's result value, and then release the lock before returning. This will allow for all results to be accessible once all threads have returned.

Similarly, if other information needs to be shared between threads for a process, the thread pool class can be further expanded upon to handle such needs.

Overall, the thread pool class allows for a user to easily write a function that could be performed in parallel.

SPEC_PARSER.PY

The file `spec_parser.py` contains the functionality responsible for accessing the specification information provided by the user in the specifications file. This script will search the current working directory for the specifications file, and if found, will return a dictionary of information corresponding to each section of the specifications file (ICUs, Patients, Parameters).

The specifications parsing is currently performed by the function `getSpecifications`. This function performs a line-by-line read through of the specifications file, keeping track of which section it is reading information for and storing it. If invalid specification values are provided, the parser will throw an error corresponding to the specification that was incorrect.

If future sections are added to the specifications file in addition to the existing three, it is very easy to add to the `getSpecifications` function. A user will only need to add a boolean value for determining if the parser is within a section, update the logic that sets the boolean values, and then add the line-based parsing logic. Lastly, the user should ensure that the data holding the parsed information is returned.

Currently, the dictionaries for each section are returned individually. This allows for the calling function to access the specifications for each section as a *n*-way tuple or individually.

DATA_ACCESS.PY

The file `data_access.py` contains the functionality responsible for setting up the SQL queries for obtaining the information from the database, executing those queries and retrieving the information, and returning that information to the calling process. It is not, however, responsible for processing this information – only the retrieval. The main function that should be called to access the specified information from Mimic is `obtainData`, which should be passed the specifications dictionaries, a connection to Mimic, and an instance of the `PatientThreadPool` class. All functionality that is meant to handle the data retrieved from the database should

be added in this function. The returned data should be unmodified from the form stored in the database, and will be passed to the data processing modules.

The function `obtainData` uses two other functions to perform its work:

- `makeQueries` uses the specification information to create the SQL queries that will fetch the data needed.
- `obtainMeasurements` adheres to the `PatientThreadPool` requirements, and is used to access patient measurement information from Mimic.

PATIENT_PROCESSING.PY

The file `patient_processing.py` is responsible for all functionality that processes the intermediate data representation into the final representation that will be used to generate the required files in the patient dataset. There currently is one function within this file, `evaluatePatients`, and it is the function responsible for processing all of the measurement data for a patient, and its output is the information that will be directly used to form the patient dataset files. This function adheres to the `PatientThreadPool` standards.

If a user/developer wishes to add future rules for handling specific measurements, this functionality should be handled within this file. For example, the specific handling of Mechanical Ventilation is performed within the function `handleMechVent`, which is located in `patient_processing.py` and is called within the function `evaluatePatients`. This function will replace the value of a mechanical ventilation event with 0.0 if there is no ventilation in use, 1.0 if ventilation is in use, and 2.0 if ventilation is ending. Another example is `handleTroponin`, which interprets inequalities used to represent very high or low Troponin measurements.

As mentioned, all processing of data should be handled by making calls to functions within this file. Although the script `data_access.py` is only responsible for data retrieval from Mimic, functions in `patient_processing.py` could be written to both retrieve data from Mimic and process it. This should only be the case if processing data requires information from Mimic specific to processing that data. For future additions, it may be best to do any processing-specific retrievals in `patient_processing.py`.

STAT_REPORT.PY

The file `stat_report.py` contains the functionality responsible for generating any reports that summarize or visualize aspects of a patient dataset. This file should be able to have its functions invoked purely on the final post-processing form of the data. This will ensure that the reports can be generated during the dataset-generation process, or afterwards in case any user alterations were provided to the dataset that require the statistics report to be re-generated.

This file contains a class, `StatReportGenerator`, that should be initialized before generating a report. By using a class, it will be easier to store and access data needed across multiple report-generation functions.

As it stands, the only input the class needs is the parameter information obtained from the function `getSpecifications` in the `spec_parser.py` module.

`StatReportGenerator` currently contains one function, `createReport`, that is used to generate a report listing the counts of measurements along with their distributions. As input, the function takes a list of patient data, either obtained from post-processing during the dataset generation or from reading through the final patient dataset files, and a directory name, for where the report should be placed. This directory name should be the same name of the directory where the patient dataset is stored.

If a user wishes to implement future functionality for report generation, additional functionality can be added via functions in this file or included within the existing function.

It is also important to note that script file can be run independently. This means that if a user wishes to further filter the results after initial dataset generation, the script `stat_report.py` can be run afterwards to generate a new report using the command below:

```
stat_report.py [directory] [specfile]
```

where `[directory]` is the directory containing the MDGL patient dataset and `[specfile]` is the specification file to use within that directory.

Setting Up and Running

This section briefly discusses how to setup the MDGL for running.

1. Make sure that Python 2.7 is installed on your machine. If you do not have Python 2.7, an installer can be found at this link: https://www.python.org/download/releases/2.7/. You can verify that python has successfully installed by opening up a command prompt, typing "`python --version`", and hitting return. This should provide you with output similar to

   ```
   Python 2.7.X
   ```

   where X denotes the specific release of Python 2.7 that you have installed.

2. Make sure that the Numpy and psycopg2 libraries are installed. If you are on a Mac or Linux machine, you can do so using the commands:

   ```
   pip install numpy
   pip install psycopg2
   ```

3. If you are having problems installing Python and/or the necessary libraries, you can instead install Python and the necessary packages using Anaconda. An installer for Anaconda can be found at this link: https://www.anaconda.com/download. Anaconda is Python distribution that makes it easy to install popular python packages used for data science. Once you have installed Anaconda, open up a terminal and use the following commands:

   ```
   conda install numpy
   conda install psycopg2
   ```

4. Next, download the MDGL code from the GitHub repository: https://github.com/wvaugha2/MimicProject. The code will be located in the folder "`mdgl`". By placing the Python scripts in your working directory, you can now begin using the MDGL using the command:

   ```
   python data_gen.py [host] [port] [specfile]
   ```

APPENDIX A

Below is an example for a specifications file that only considers patients who have at least 48 hours within an ICU stay of type CCU, SICU, MICU, and CSRU; are above the age of 16; and are either male or female:

```
#ICUs

CCU     True
SICU    True
MICU    True
NICU    False
CSRU    True
TSICU   False

#Patients

Age; 16; 18
Sex; Both
Hours; 24

#Parameters

Albumin; ALBUMIN; [50862,1521,226981]
ALP; ALKALINE_PHOSPHATASE; [50863,3728]
ALT; ALANINE_TRANSAMINASE; [769,220644]
AST; ASPARTATE_TRANSAMINASE; [50878,220587,3801]
Bilirubin; BILIRUBIN; [50885,225690]
BUN; BLOOD_UREA_NITROGEN; [51006,1162,225624]
Cholesterol; CHOLESTEROL; [50907,789,1524,220603,3748]
Creatinine; CREATININE; [50912,51081,227005,1525,220615]
DiasABP; INVASIVE_DIASTOLIC_ARTERIAL_BLOOD_PRESSURE; [225310,8368,220051,8555,8364]
FiO2; FRACTIONAL_INSPIRED_O2; [190,223835,3420]
GCS; GLASGOW_COMA_SCORE; [198,226755]
Glucose; SERUM_GLUCOSE; [50809,50931,227015,3744,1529]
HCO3; SERUM_BICARBONATE; [227443,226759,812]
HCT; HEMATOCRIT; [51480,51221,227017,813]
HR; HEART_RATE; [220045,211]
K; SERUM_POTASSIUM; [50971,227442,1535]
Lactate; LACTATE; [50813,1531,225668]
Mg; SERUM_MAGNESIUM; [50960,1532,220635]
MAP; INVASIVE_MEAN_ARTERIAL_BLOOD_PRESSURE; [224, 224322]
MechVent; MECHANICAL_VENTILATION; [467,468,720,722]
Na; SERUM_SODIUM; [50983,1536,220645]
NIDiasABP; NONINVASIVE_DIASTOLIC_ARTERIAL_BLOOD_PRESSURE; [220180,8441]
NIMap; NONINVASIVE_MEAN_ARTERIAL_BLOOD_PRESSURE; [220052,220181]
NISysABP; NONINVASIVE_SYSTOLIC_ARTERIAL_BLOOD_PRESSURE; [220179,455]
PaCO2; PARTIAL_PRESSURE_OF_ARTERIAL_CO2; [778]
PaO2; PARTIAL_PRESSURE_OF_ARTERIAL_O2; [779]
ph; ARTERIAL_pH; [780,50831,50820,223830]
Platelets; PLATELETS; [51265,828,227457]
RespRate; RESPIRATION_RATE; [618,3603,220210]
SaO2; O2_SATURATION_IN_HEMOGLOBIN; [50817,220227]
SysABP; INVASIVE_SYSTOLIC_ARTERIAL_BLOOD_PRESSURE; [225309,51,220050]
Temp; TEMPERATURE; [50825,3655,677,223762,676]
TropI; TROPONIN-I; [51002]
TropT; TROPONIN-T; [51003,227429]
Urine; URINE; 51108,40055,43175,40069,40094,40715,40473,40085,40057,40056,40405,40428,40086,
40096,40651,226559,226560,227510,226561,226584,226563,226564,226565,226567,226557,226558]
WBC; WHITE_BLOOD_CELL_COUNT; [51301,51300,220546,1542]
Weight; WEIGHT; [763, 224639]
```